PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMA DE DETECCIÓN Y EVASIÓN DE OBSTÁCULOS POR MEDIO DE UN LIDAR 360° PARA UN SISTEMA AÉREO NO TRIPULADO

Tesis para obtener el título profesional de Ingeniero Electrónico

AUTOR:

Alfredo Leonardo Chávez Cobián

ASESOR:

Mg. Ing. Carlos Saito Villanueva

Lima, setiembre, 2020

RESUMEN

Los sistemas aéreos no tripulados (SANT) se han convertido en una comodidad asequible para cualquier fin. Sin embargo, estos dispositivos pueden causar daño tanto a infraestructuras como a personas en caso de colisión. De esta manera, el problema radica en que no existe en el Grupo de Investigación de Sistemas Aéreos No Tripulados un módulo electrónico capaz de detectar y evadir obstáculos en escenarios tales como bordear una estructura fija o evitar una colisión inminente con algún objeto que se interponga entre el SANT y la meta. Además, que se pueda acoplar a diversas plataformas y que tenga un rango de detección de 360°.

La solución al problema mencionado se llevó a cabo mediante el desarrollo de un sistema de detección y evasión de obstáculos. En cuanto al hardware, se eligió como sensor al Sweep LiDAR 360°, al Odroid C2 como computadora acompañante y al Pixhawk como controlador de vuelo. La plataforma elegida fue el cuadricóptero Tarot FY450. En cuanto al Software, se diseñó un algoritmo de adaptación de rutas basado en 4 modos de vuelo.

El flujo de información da inicio con la adquisición de datos del entorno por parte del sensor LiDAR. Dicha información es ordenada del punto más cercano al más lejano y posteriormente es filtrada en base a la intensidad de señal. La información resultante es procesada en la computadora acompañante y un modo de vuelo es elegido en base a criterios previamente establecidos.

En cuanto a las pruebas realizadas para comprobar la eficiencia del sistema, se realizaron simulaciones en Matlab y pruebas reales. En cuanto a las pruebas reales, se realizaron 3 con un biombo y una con una pancarta. El objetivo de las 3 primeras pruebas fue evaluar el dispositivo en un entorno controlado, mientras que la prueba con pancarta tuvo como objetivo evidenciar el modo de vuelo de emergencia (Avoid Obstacle). Además, el límite de velocidad resultante fue de 0.5 m/s.

DEDICATORIA

A mis padres, por su apoyo incondicional en cada etapa de mi formación, por haber depositado la confianza que hizo posible este trabajo

A mi hermano, para que sea capaz de seguir una vocación que lo llene de satisfacción en cada instante de su vida.

AGRADECIMIENTO

A mis padres, por motivarme constantemente.

A mis maestros, por incentivar la curiosidad que hizo posible este trabajo.

A mi asesor, por brindarme la oportunidad de formar parte del proyecto que hizo posible este trabajo.

A ella, quien me brindó apoyo y comprensión cuando más lo necesitaba.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Los sistemas aéreos no tripulados (SANT[1]) se han convertido en una comodidad asequible para cualquier fin. Tal es el caso de la fotografía de precisión, investigación aeronáutica y fines recreativos. Sin embargo, estos dispositivos pueden causar daño tanto a infraestructuras como a personas en caso de colisión. Los accidentes reportados a causa de la imprudencia al pilotar un SANT han llevado a plantear restricciones contra estos.

No existe en el Grupo de Investigación de Sistemas Aéreos No Tripulados un módulo electrónico capaz de detectar y evadir obstáculos en escenarios tales como bordear una estructura fija o evitar una colisión inminente con algún objeto que se interponga entre el SANT y la meta. Además, que se pueda acoplar a diversas plataformas y que tenga un rango de detección de 360°.

La detección y evasión de obstáculos para SANT tiene diversos enfoques. Estos se basan principalmente en técnicas de visión por computadora, análisis de imágenes o fusión de sensores capaces de medir distancia. En el caso de la fusión de sensores, se construyen arreglos de sensores ultrasónicos o infrarrojos para determinar la mínima distancia respecto a un objeto en cualquier dirección. Esta información es posteriormente utilizada en un algoritmo de adaptación de rutas.

Con el fin de prevenir accidentes y la pérdida de activos costosos, la presente tesis propone: desarrollar un módulo electrónico capaz de detectar y evadir obstáculos en escenarios tales como bordear una estructura fija o evitar una colisión inminente con algún objeto que se interponga entre el SANT y la meta. Además, que se pueda acoplar a diversas plataformas y que tenga un rango de detección de 360°. Dicho módulo electrónico constará de una computadora acompañante capaz de interactuar por medio de comandos con un controlador de vuelo Pixhawk. De esta manera, la trayectoria inicial podrá ser

[1] En adelante los nemónicos entre paréntesis reemplazarán los nombres a los que están relacionados.

corregida evadiendo así los obstáculos detectados por el sensor o arreglo de sensores.

En el primer capítulo se define la problemática y estado del arte. En base a esto, se plantean el objetivo general y los objetivos específicos. Por último, se desarrolla el marco conceptual.

En el segundo capítulo, se analiza el hardware a utilizar sustentado mediante cuadros comparativos. Además, se presentan las conexiones entre los dispositivos seleccionados y el diseño de la estructura que agrupa el sistema en sí.

En el tercer capítulo, se desarrolla el algoritmo de adaptación de rutas por medio de la detección y evasión de obstáculos. Este será programado en Python. El algoritmo estará compuesto de cuatro modos de vuelo los cuales serán verificados en Matlab.

Por último, en el cuarto capítulo, se ejecutan las pruebas del sistema y se delimita la velocidad máxima de operación para asegurar la respuesta óptima del sistema. Finalmente, se presentan los resultados, se detallan las conclusiones y se mencionan las recomendaciones respectivas.

CAPÍTULO 1

SISTEMA DE DETECCIÓN Y EVASIÓN DE OBSTÁCULOS

En este capítulo se explicará el sistema de detección y evasión de obstáculos planteado. Para ello, se definirá adecuadamente el problema, se expondrán trabajos relacionados al tema en el estado del arte y basado en ello se plantearán el objetivo general y los objetivos específicos. Finalmente, se definirá el marco conceptual.

1.1 Definición del problema

El consumo de sistemas aéreos no tripulados, de ahora en adelante denominado SANT, tanto para uso personal como para uso comercial se encuentra en aumento. Se hizo público un estudio en el cual se estimó que en 2017 se iban a vender 3 millones de SANT destinados tanto para uso personal como comercial [1].

En el territorio peruano, el uso de SANT se encuentra regulado por la Dirección General de Aeronáutica Civil (DGAC). Este organismo expide las licencias de operador de SANT, por ende, limita los posibles daños ocasionados al operar las aeronaves de manera ilícita. Sin embargo, la

locación en la cual se realizan los vuelos, las construcciones circundantes o el terreno accidentado o montañoso aumenta la posibilidad de una colisión.

En la actualidad, el uso de SANT es ampliamente difundido para labores de filmación aérea, modelamiento en 3D mediante fotogrametría, cálculo de volúmenes, inspección visual en minería, construcción, etc. BBVA en su artículo ¿Quien lidera el mercado de los Drones? menciona que las ventas de SANT alcanzarán los 12.000 millones en 2021. Esto significa una tasa de crecimiento anual compuesta (CAGR) del 7,6%, pasando de los 8.500 millones en 2016 [2].

Empresas como Aereal en España, utilizan SANT industriales de tipo multirotor tales como el Matrice 200 y Matrice 600 de la empresa DJI. Dicha empresa se encuentra habilitada por la Agencia Estatal de Seguridad Aérea para manejo de SANT la cual es análoga a la DGAC en el territorio peruano [3].

A diferencia de los SANT anteriormente mencionados, SANT tales como el Phantom 4 de la empresa DJI y Typhoon H de la empresa Yuneec poseen la función de detección y evasión de obstáculos gracias a los sensores montados alrededor de la carcasa. Sin embargo, la menor autonomía y tamaño limita la cantidad de funciones que pueden realizar a nivel industrial. En adición, el sistema de detección y evasión de obstáculos está limitado únicamente a dicha plataforma y no todos tienen un rango de detección de 360°.

La Universidad RMIT, publicó un estudio en el cual se evaluaron 150 incidentes civiles reportados ocurridos durante 2006-2016. El estudio indica que el 64% de los incidentes fueron ocasionados debido a problemas técnicos. Además, se descubrió que, en la mayoría de casos, los enlaces de comunicación rotos entre el piloto y el SANT fueron la causa del accidente [4].

Frente a este contexto, el problema es que El Grupo de Investigación de Sistemas Aéreos No Tripulados no cuenta con un sistema que sea capaz de

detectar y evadir obstáculos en escenarios tales como bordear una estructura fija o evitar una colisión inminente con algún objeto que se interponga entre el SANT y la meta. Además, que se pueda acoplar a diversas plataformas y que tenga un rango de detección de 360°.

1.2 Estado del arte

En cuanto a la detección y evasión de obstáculos se refiere, los intentos por detectar obstáculos han tenido dos soluciones ampliamente usadas. La primera hace uso de diversos sensores para poder brindar autonomía al SANT, y la segunda analiza el entorno a través del lente de una cámara y luego del respectivo análisis se logra la detección de los obstáculos. Existe una tercera solución la cual consta de hacer uso de las dos soluciones anteriores para lograr el mismo objetivo. Todas ellas tienen puntos a favor y en contra.

Por un lado, el uso de sensores brinda datos relativamente precisos del entorno y permite procesar rápidamente la información. Claro está, dependiendo del tipo de sensor y del volumen de información que éste recopile.

En cuanto al uso de sensores ultrasónicos, Rusyadi y Seong acoplaron a un SANT de tipo multirotor un arreglo de 3 sensores de ultrasonido. Dichos sensores tienen como finalidad recopilar información del entorno y enviarla a la computadora acompañante. Este estudio fue desarrollado mediante un API (Interfaz de Programación de Aplicaciones) y fue pensado para compensar la deficiencia que existe al verse limitado el uso de GPS (Sistema de Posicionamiento Global) en interiores. En contraposición, la lectura del arreglo de sensores mostró ser deficiente en presencia de superficies blandas como materiales de espuma. [5]

Pandey desarrolla lo que denomina “Gorra” la cual consta de un arreglo de sensores de ultrasonido en 360° dentro de una estructura moldeada. Dicho

arreglo es montado en la parte superior del SANT y, en comparación con el ejemplo anteriormente expuesto, busca cubrir la mayor área posible de detección [6]. Es importante mencionar que además de la deficiencia anteriormente mencionada, la detección por ultrasonido sólo es efectiva en cierto rango.

Los sensores LiDAR (Detección de luz y rango) han ganado bastante presencia en el campo de la automatización. Actualmente son ampliamente difundidos en el sector automovilístico.

La Administración Nacional Oceánica y Atmosférica (NOAA) define a los LiDAR como topográficos y batimétricos: “El LIDAR topográfico generalmente usa un láser infrarrojo cercano para mapear la tierra, mientras que el LiDAR batimétrico usa luz verde que penetra el agua para medir también las elevaciones del lecho marino y del lecho del río” [7].

Ibáñez menciona que típicamente el LiDAR es usado para dos tópicos: mapeo en entornos cerrados y mapeo de entorno/obstáculo en exteriores. En ambos casos, se sigue usando multirotores para poder montar el sistema mencionado pues brindan la estabilidad necesaria durante la creación de mapas tanto en 2D como en 3D [8].

En cuanto al uso del LiDAR en SANT, Sabatini y Gardi hacen uso de un escáner láser que tiene un ángulo de visión de 40° en azimut y 30° en elevación, con un margen de campo de $\pm 20°$ para la detección de líneas de transmisión eléctrica [9].

Finalmente, en el intento por mejorar la capacidad de detección de obstáculos se ha llegado a intentar cubrir todos los ángulos posibles mediante un arreglo de sensores estáticos, el cual usa 16 sensores infrarrojos y 12 sensores ultrasónicos para conseguir una redundancia de 360°. [10]

En cuanto al algoritmo de adaptación de rutas, Adouane hace uso de un algoritmo de evasión de obstáculos orbital. Dicho trabajo brinda solución a la

navegación de robots en espacios confinados. Este algoritmo tiene como concepto primordial la atracción y repulsión de campos potenciales artificiales. De esta manera, el robot se desplaza a través de un espacio confinado bajo la acción de fuerzas de atracción y repulsión las cuales delimitan la ruta a seguir. [11]

El algoritmo de evasión de obstáculos orbital hace uso de un planeamiento de rutas reactivo (es decir, no existe una ruta trazada en concreto). Por ello, la trayectoria del robot varía en tiempo real con cada obstáculo detectado. Otro tipo de algoritmo de adaptación de rutas se centra en el seguimiento de paredes para llegar a la meta. En [12] se realiza el seguimiento de la pared de un túnel por medio de ultrasonidos ensamblados en un cuadricóptero.

El algoritmo utilizado para mantener el SANT en la posición deseada sin necesidad de utilizar GPS es PID. Las constantes en tiempo continuo de ganancia proporcional (Kp), ganancia integral (Ki) y ganancia derivativa (Kd) fueron halladas mediante depuración. La posición a lo largo del tiempo es hallada mediante la fusión de las coordenadas teóricas y las coordenadas actuales obtenida por el arreglo de sensores ultrasónicos.

P. Raja y S. Pugazhenti dividen los algoritmos de adaptación de rutas en: fuera de línea y en línea. Los algoritmos fuera de línea comprenden el escenario en el cual los obstáculos son estacionarios y el entorno se conoce a cabalidad. De esta manera la ruta a seguir se encuentra preestablecida. Los algoritmos en línea comprenden el escenario en el cual la información del entorno no se encuentra disponible. En este caso el SANT recopila información por medio de sensores mientras se desplaza a través del entorno. [13]

La navegación en línea puede ser esquematizada por medio de un flujo a través de 4 bloques: bloque de cognición y planeamiento de ruta, bloque de control de movimiento, bloque de percepción y bloque de localización y construcción de mapas. El flujo inicia cuando la misión (meta) ingresa al primer bloque, como reultado se obtiene una ruta a seguir. Dicha ruta ingresa al segundo bloque el cual será probado en el entorno. En esta etapa se hace

uso del tercer bloque, los sensores instalados en el SANT adquieren información del entorno y brindan data sin procesar para el último bloque. Finalmente, el último bloque brinda la posición del SANT y una aproximación del entorno mediante un mapa.

Existen dos enfoques en cuanto a los algoritmos de adaptación de rutas se refiere: enfoque clásico y enfoque evolutivo. En cuanto al enfoque clásico se puede mencionar el método de descomposición celular [14], gráficos de visibilidad y driagramas Voronoi [15]. En cuanto el enfoque evolutivo se puede mencionar el uso de Algoritmos Genéticos [16] o Algoritmos de Optimización por Enjambre de Partículas [17].

1.3 Marco Conceptual

En este punto se definen cada uno de los dispositivos a utilizarse durante la implementación de la presente tesis. En primera instancia se justificará el uso del cuadricóptero FY450 como plataforma mediante un cuadro comparativo. En cuanto al sensor, se justificará la elección del LiDAR Sweep 360° comparándola con otras opciones en el mercado. Por último, la computadora acompañante será evaluada en función de su número de puertos, memoria, procesador y conectividad.

1.3.1 Elección de la plataforma de vuelo

Si bien en la última década los sistemas aéreos no tripulados han crecido aceleradamente, estos se vienen desarrollando desde inicios del siglo XX. En [18] se indica que los sistemas aéreos no tripulados están básicamente constituidos por un segmento aéreo y un segmento en tierra.

Una vez introducido el concepto de SANT, es necesario centrarse en aquellos que representan una alternativa para la plataforma de vuelo en la cual se

montará el sistema de detección y evasión de obstáculos. Los criterios para la elección de la plataforma adecuada son: la estabilidad, tamaño adecuado para montar el sensor y computadora acompañante, robusto, de fácil adquisición y de bajo costo. De esta manera, aquellos SANT de ala rotatoria (multirotores) son la mejor opción. Por este motivo, se realizará un cuadro comparativo entre las distintas opciones de multirotores.

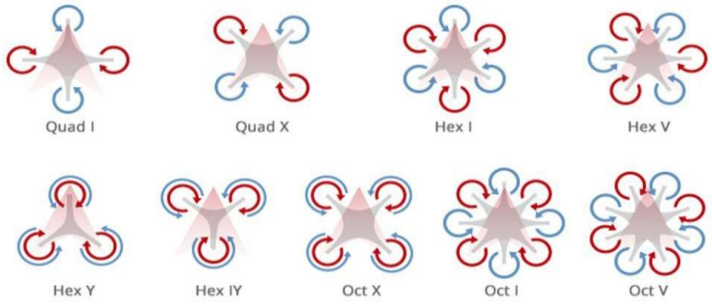

Figura 1. Clasificación de Multirotores [19]

Aerpas, la asociación española de RPAS (Sistema Aéreo Pilotado Remotamente), clasifica a los SANT según la cantidad de motores en: tricópteros (3 motores), cuadricópteros (4 motores), hexacópteros (6 motores) y octocópteros (8 motores). Esto se puede apreciar en la figura 1. En cuanto a la estabilidad, a mayor número de motores, mayor será la estabilidad, pero mayor el consumo. De esta manera, los tricópteros son poco estables y los octocópteros consumen demasiado.

Tabla 1. Comparación de plataformas de vuelo

	SANT 4 MOTORES	SANT 6 MOTORES	SANT 8 MOTORES
Imágen Referencial			
Estabilidad	Media	Alta	Muy Alta
Consumo	Medio	Alto	Muy Alto
Complejidad	Baja	Media	Alta
Carga Útil en gramos	600 ~ 800	1000 ~ 1200	1600 ~ 6600
Precio en dólares	239	369	1500

(Fuente: Elaboración propia)

De acuerdo con la tabla 1, la plataforma elegida será el cuadricóptero. Específicamente el modelo Tarot FY450. Sobre ésta, se instalará una plataforma adicional diseñada para montar al sensor y la computadora acompañante.

1.3.2 Elección de la computadora acompañante

El procesamiento de la información adquirida por el sensor es esencial para la detección de obstáculos. Por este motivo la computadora acompañante debe poseer gran velocidad de procesamiento, múltiples puertos USB, tener un entorno fácil programación, ser de bajo costo y de fácil adquisición. Frente a estos requerimientos, se han evaluado cuatro posibles opciones para determinar cuál de ellas es la más adecuada.

Tabla 2. Comparación de computadoras acompañantes

	RASPBERRY PI 3	ODROID-C2	ODROID-XU4	BEAGLEBONE BLACK
Imágen Referencial				
CPU	CPU A 1.2 GHz 64-bit quad-core ARM v8	Amlogic ARM° Cortex°-A53(ARMv8) 1.5GHz quad-core CPUs	Sams. Exynos5422 Cortex°- A15 2GHz & Cortex°-A7 Octa core CPUs	AM335x-1GHz ARM° Cortex-A8
RAM	1 GB	2 GB DDR3 SDRAM	2 GB LPDDR3 RAM PoP stacked	512 MB DDR3 RAM
Número de puertos USB	4	4	3	1
Número de pines	40 (GPIO)	40 (GPIO) 7(I2S)	30 (GPIO) 12(I2S)	92 (GPIO)
Sistema Operativo	RASPBIAN	Ubuntu 16.04 o Android 5.1 Lollipop basado en Kernel 3.14 LTS	Linux Kernel 4.9 LTS	DEBIAN
Precio en dólares	35	46	59	45

(Fuente: Elaboración propia)

Como se aprecia en la tabla 2, la opción más viable es el Odroid C2 debido a que posee cuatro puertos USB, tiene la mayor memoria RAM de las cuatro opciones, posee un entorno de programación sencillo de usar (Ubuntu), y tiene un precio adecuado. De esta manera, la computadora acompañante elegida será el vínculo entre el sensor LiDAR y el controlador de vuelo Pixhawk.

1.3.3 Elección del sensor

En cuanto a la elección del sensor, se ha expuesto las fortalezas y debilidades de los distintos métodos de detección de obstáculos en el estado del arte de la presente tesis. Por ese motivo, la elección del sensor se verá limitada a elegir el LiDAR idóneo para el sistema de detección y evasión de obstáculos planteado.

Tabla 3. Comparación entre sensores LiDAR presentes en el mercado.

	RPLiDAR A1	RPLiDAR A2	SWEEP LiDAR
Imágen Referencial			
Número de muestras/Seg	2000	4000	1000
Alcance	< 12 m	< 16 m	< 40 m
Operación de acuerdo al entorno	Puede ser dañado por contacto directo con el sol	No recomendable con luz solar directa	Luz solar completa
Peso	190 g	190 g	120 g
Frecuencia de rotación	< 10	< 10	< 10
Precio en Dólares	399	449	349

(Fuente: Elaboración propia)

Como se observa en la Tabla 3, la opción más viable debido a las cualidades, procesamiento relativamente sencillo, confiabilidad de las lecturas, ángulo de visión en 360°, reducido tamaño y peso es el sensor Sweep LiDAR.

1.3.4 Lenguaje de programación Python y protocolo de comunicación MAVLink

El lenguaje de programación Python y el protocolo de comunicación MAVLink son herramientas imprescindibles para la correcta ejecución del sistema de detección y evasión de obstáculos.

Lenguaje de Programación Python (versión 3.6):

Es un lenguaje de programación orientado a objetos, posee cerca de 20 000 librerías las cuales van aumentando gracias a la gran cantidad de personas que aportan constantemente en una comunidad activa. La codificación es similar al pseudo código lo cual permite que su aprendizaje sea en gran medida intuitivo. Se usa un intérprete en lugar de un compilador, esto permite que los errores sean detectados al “correr” el algoritmo. [20]

Protocolo de comunicación MAVLink (versión 2):

Tal y como se expone en [21], MAVLink es un protocolo de comunicación ligero orientado a la comunicación con SANT y entre componentes al interior de un SANT. Los mensajes enviados constan de archivos en formato XML, el cual es un lenguaje de marcado extensible. [22]

1.4 Objetivo General y Objetivos Específicos

Objetivo General

Desarrollar un módulo electrónico capaz de detectar y evitar la colisión del SANT con obstáculos durante el vuelo en escenarios tales como bordear una estructura fija o evitar una colisión inminente con algún objeto que se interponga entre el SANT y la meta. Además, que se pueda acoplar a diversas plataformas y que tenga un rango de detección de 360°. Esto, con el fin de prevenir accidentes además de la pérdida de activos costosos.

Objetivos Específicos

- Elegir el sensor o arreglo de sensores que permitan obtener información del entorno. Además, seleccionar la computadora acompañante que procese dicha información.
- Diseñar una estructura compacta y ligera capaz de contener los dispositivos y permitir que el proceso de ensamblaje al SANT sea sencillo.
- Desarrollar un algoritmo de adaptación de rutas que permita a la computadora acompañante procesar la información procedente del sensor o arreglo de sensores. Posteriormente, enviar el comando de vuelo hacia el Pixhawk a través del protocolo de comunicación Mavlink.
- Llevar a cabo las pruebas controladas necesarias con la finalidad de demostrar el desempeño del Sistema de detección y evasión de obstáculos (SDEO). Además, evaluar las limitaciones del sistema desarrollado.

CAPÍTULO 2

DESARROLLO DEL HARDWARE DEL SISTEMA DE DETECCIÓN Y EVASIÓN DE OBSTÁCULOS

Este capítulo se expondrá el marco teórico, posteriormente se evaluará la metodología a seguir para el desarrollo de la presente tesis. Se mostrará el diagrama de conexiones eléctricas entre dispositivos. Finalmente, se expondrá el diseño de la estructura que anexa los componentes del sistema de detección y evasión de obstáculos a la plataforma del SANT.

2.1 Marco Teórico

La comunicación entre los dispositivos que conforman el Sistema de Detección y Evasión de Obstáculos, de ahora en adelante denominado SDEO, es fundamental. Por este motivo, la Figura 2 muestra el flujo de información por etapas entre los dispositivos que conforman el SDEO y el entorno.

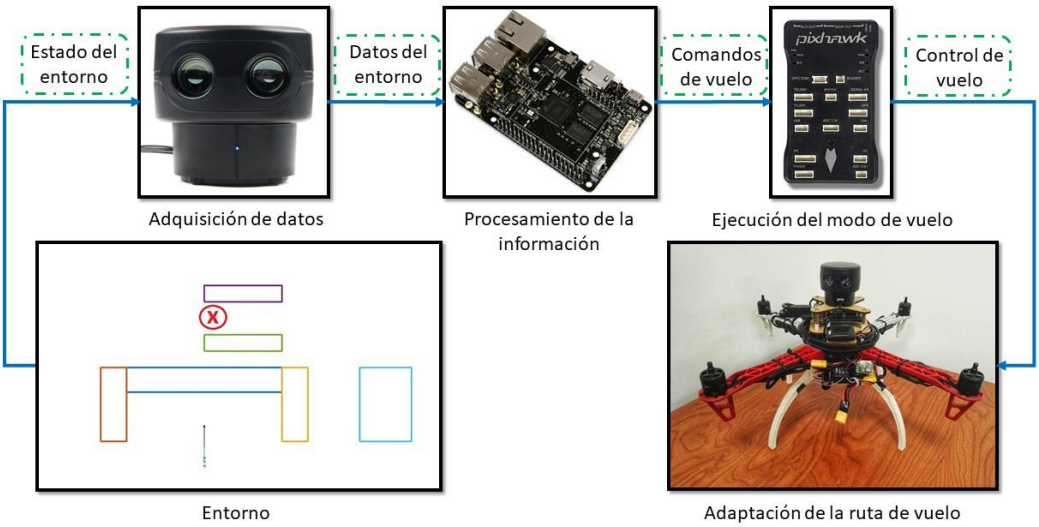

Figura 2. Etapas del proceso de detección y evasión de obstáculos (Fuente: Elaboración propia)

Se puede apreciar en la Figura 2 un entorno simulado en Matlab. Dicho entorno es una representación de lo esperado durante la etapa de pruebas. El círculo rojo con una equis en el interior representa el punto de llegada o meta. A continuación, se profundizará en cada una de las etapas.

Adquisición de datos:

El primer componente del SDEO es el sensor LiDAR y representa la etapa de adquisición de datos provenientes del entorno. De esta manera el sensor capta el estado del entorno mediante un bloque de datos de 7 bytes con la siguiente estructura: [23]

- Sincronismo / Error (1 byte): El bit menos significativo es el bit de sincronismo, mientras que los 7 bits más significativos indican error en la comunicación.
- Grados azimut (2 bytes): Indica el ángulo en el que fue recopilada la información.
- Distancia en centímetros (2 bytes): Distancia al punto medido.
- Intensidad de señal (1byte): Intensidad de señal de la medición en un rango de 0 – 255.
- Suma de control (1 byte): Calculada añadiendo los 6 bytes de datos, dividiendo entre 255 y manteniendo el remanente.

Procesamiento de la información:

En esta etapa, la información recopilada por el sensor es procesada por el algoritmo de detección y evasión de obstáculos, de ahora en adelante denominado ADEO. Dicho algoritmo ordena la información, la filtra y posteriormente asigna la acción correspondiente mediante un comando de vuelo.

Ejecución del modo de vuelo:

En esta etapa, el controlador de vuelo Pixhawk recibe el comando de vuelo y lo ejecuta.

Adaptación de la ruta de vuelo:

Finalmente, en esta etapa, la ejecución del comando de vuelo se realiza mediante la variación de la velocidad en los motores del SANT por medio del variador de velocidad electrónico (ESC).

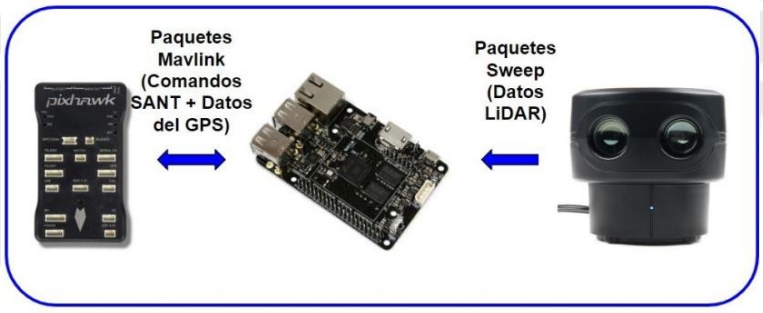

Figura 3. Envío de información del Sistema de detección y evasión de obstáculos (Fuente: Elaboración propia)

Puede apreciarse en la Figura 3 que el sensor envía información unidireccional a la computadora acompañante. Sin embargo, la computadora acompañante comparte un flujo de información bidireccional con el controlador de vuelo. Dicha comunicación se realiza por medio del protocolo de comunicación MAVLink, es un protocolo de comunicación ligero para comunicación con SANT y entre componentes de SANT a bordo.

2.2 Metodología

En la primera etapa del desarrollo de la presente tesis, el desarrollo se enfocará en función de los dos objetivos específicos presentados a continuación:

- Elegir el sensor o arreglo de sensores que permitan obtener información del entorno. Además, seleccionar la computadora acompañante que procese dicha información.
- Diseñar una estructura compacta y ligera capaz de contener los dispositivos y permitir que el proceso de ensamblaje al SANT sea sencillo.

Para dicho fin es necesario adquirir todos los elementos que componen el SANT. Entre estos elementos se encuentran los motores, los controladores de velocidad electrónicos, el receptor de datos, los cables, los conectores, la batería y todos los elementos evaluados en el Capítulo 1.

La Tabla 4 enumera las tareas a realizar previas a la adquisición de componentes y aquellas que permitan el cumplimiento de los dos objetivos específicos anteriormente expuestos.

Es importante recalcar que el periodo asignado a las tareas no indica que estas deban realizarse una a continuación de la otra. Únicamente indica cual es la duración de cada una. Sin embargo, las tareas pueden realizarse en simultáneo.

Tabla 4. Lista de tareas asignadas para la primera etapa (Fuente: Elaboración propia)

	Tarea	Periodo (en semanas)
1	Investigación del tema elegido	18 Semanas
2	Desarrollo de la problemática	2 semanas
3	Definición de los objetivos	2 semanas
4	Elección de componentes	1 semana
5	Adquisición de componentes	4 Semanas

6	Diseño de estructura	4 Semanas
7	Ensamblaje de estructura	2 Semanas
8	Conexión de componentes	1 Semana

En la segunda etapa del desarrollo de la presente tesis, el desarrollo se enfocará en función de los dos objetivos específicos presentados a continuación:

- Desarrollar un algoritmo de adaptación de rutas que permita a la computadora acompañante procesar la información procedente del sensor o arreglo de sensores. Posteriormente, enviar el comando de vuelo hacia el Pixhawk a través del protocolo de comunicación Mavlink.
- Llevar a cabo las pruebas controladas necesarias con la finalidad de demostrar la eficiencia y evaluar las limitaciones del sistema desarrollado.

Para dicho fin es necesario desarrollar una estrategia de solución la cual permita elaborar el algoritmo de adaptación de rutas tomando en cuenta el cumplimiento de los dos objetivos específicos anteriormente mencionados.

La Tabla 5 enumera las tareas a realizar para lograr los objetivos específicos anteriormente descritos.

Es importante recalcar que el periodo asignado a las tareas no indica que estas deban realizarse una a continuación de la otra. Únicamente indica cual es la duración de cada una. Sin embargo, las tareas pueden realizarse en simultáneo.

Tabla 5. Lista de tareas asignadas para la segunda etapa (Fuente: Elaboración propia)

	Tarea	Periodo (en semanas)
1	Desarrollo de la estrategia de solución la cual permita elaborar el algoritmo de adaptación de rutas	5 Semanas

2	Investigación sobre algoritmos de evasión de obstáculos	18 Semanas
3	Desarrollo del algoritmo	15 Semanas
4	Pruebas de adquisición de datos con el sensor	1 Semana
5	Pruebas de comunicación mediante MAVLink	2 Semanas
6	Elaboración de algoritmo en Matlab	5 Semanas
7	Pruebas del algoritmo en Matlab	1 Semana
8	Definición de los modos de vuelo	1 Semana
9	Ejecución de pruebas con una pancarta y un biombo	3 Semanas
10	Prueba de los modos de vuelo	4 Semanas
11	Recopilación de datos	4 Semanas
12	Análisis de resultados	4 Semanas

Es necesario ahondar en la estrategia de solución del algoritmo de detección y evasión de obstáculos pues supone el grueso del desarrollo de la presente tesis. De esta manera, la estrategia a seguir puede observarse en la Figura 4.

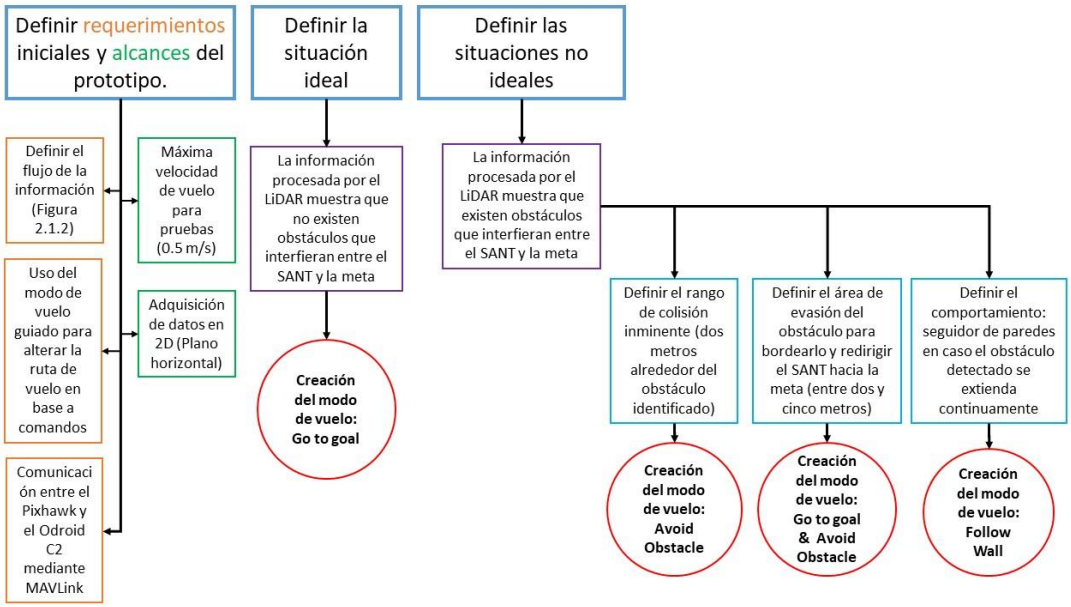

Figura 4. Estrategia de solución del algoritmo de detección y evasión de obstáculos (Fuente: Elaboración propia)

2.3 Diagrama de conexiones eléctricas entre dispositivos

En cuanto a las conexiones del SDEO, la presente tesis se enfocará en las conexiones entre el sensor LiDAR, la computadora acompañante y el controlador de vuelo. Sin embargo, no se ahondará en las conexiones entre el controlador de vuelo y los demás componentes del SANT.

Puede apreciarse en la Figura 5 la conexión entre los componentes del SDEO. El controlador de vuelo Pixhawk se encuentra fijado a la plataforma de vuelo Tarot FY450. La alimentación de la plataforma de vuelo y la computadora acompañante es por medio de una batería LiPo Multistar (Litio - Polímero) de 14.8 voltios, 4 celdas, 6600mAh. La alimentación se bifurca entre el plato conductor que alimenta a todos los componentes del SANT y la alimentación de la computadora acompañante por medio del puerto DC Jack de 5 voltios.

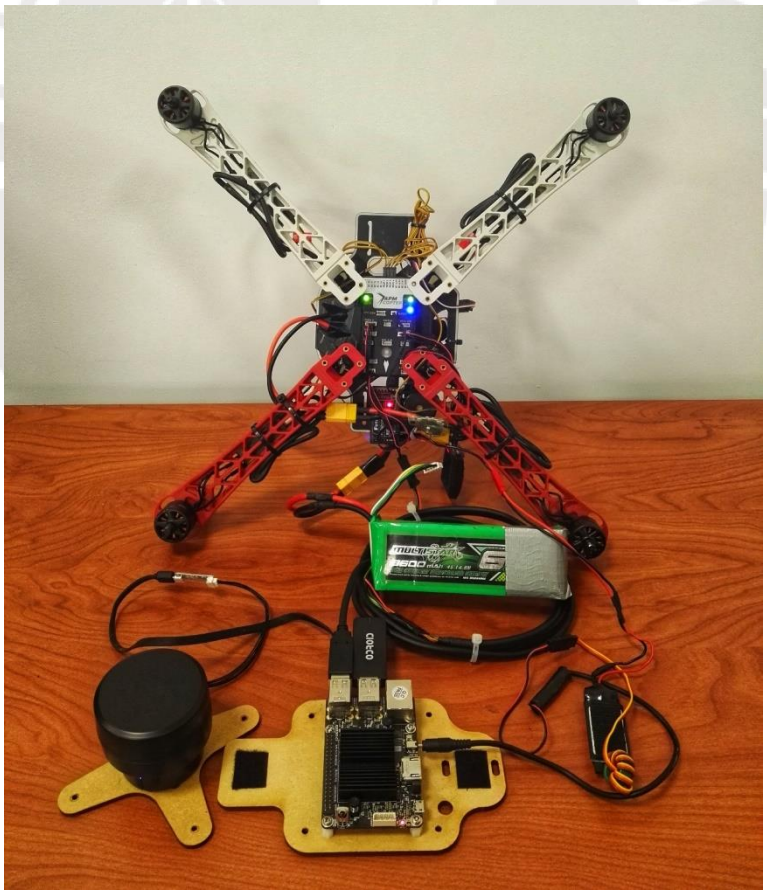

Figura 5. Conexiones eléctricas entre componentes del SDEO (Fuente: Imagen Propia)

En la Figura 6, se observa un Regulador Switch para fijar el voltaje de alimentación de la computadora acompañante en 5V y contribuir con la eliminación de ruido. La bifurcación destinada a la alimentación de la computadora acompañante tiene como etapa intermedia dicho Regulador. De esta manera se eliminan posibles fallas en la ejecución del ADEO debido a ruido en la alimentación.

Se aprecia en la Figura 5 que se están utilizando 3 puertos USB de la computadora acompañante.
El primer puerto está destinado a la conexión entre el sensor LiDAR Sweep 360° y la computadora acompañante mediante un cable conector 6 pines - USB.

Figura 6. Regulador Switch para eliminación de ruido (Fuente: Imagen Propia)

El segundo puerto USB está destinado al módulo de Wi-Fi del Odroid C2, Odroid C2 - Dongle.
El tercer puerto USB está destinado a la conexión con el controlador de vuelo. Dicha conexión se realiza mediante la adaptación del conector de telemetría 2 del Pixhawk hacia el puerto USB del Odroid C2. Finalmente, el cuarto puerto USB queda libre para la conexión de periféricos durante la etapa de pruebas.

En cuanto a la adaptación del puerto de telemetría 2 del controlador de vuelo Pixhawk hacia el puerto USB de la computadora acompañante Odroid C2 se realizó como se observa en la Figura 7.

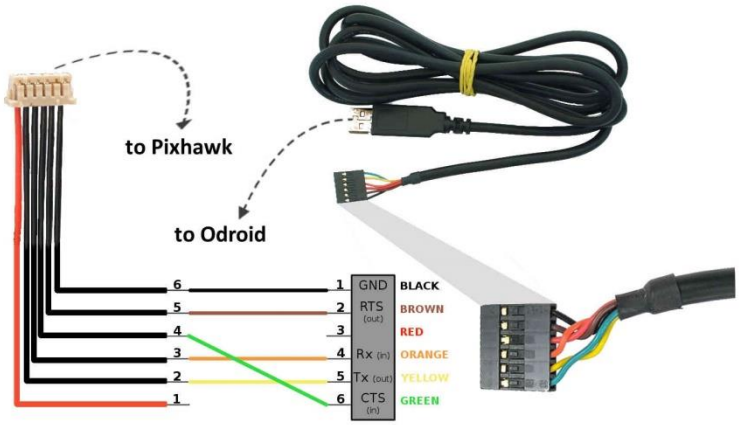

Figura 7. Conexión FTDI (Pixhawk) - USB (ODROID C2) [24]

2.4 Diseño de estructura para el ensamblaje del sensor y la computadora acompañante

En cuanto a la estructura destinada al acople de los componentes del SDEO y la plataforma de vuelo, la estructura ha sido diseñada en Inventor, elaborada en MDF e impresa en 3D.

La estructura está pensada para acoplarse a la placa superior de la plataforma de vuelo Tarot FY450. Los niveles de la estructura están delimitados por espaciadores. De esta manera, la computadora acompañante y el sensor poseen espacio suficiente para organizar el cableado de conexiones y este no interfiera durante el vuelo con las hélices del SANT.

En la Figura 8 puede observarse con detalle los niveles de la estructura de acople y la organización de cableado alrededor de esta. Es importante recalcar que el diseño de la estructura permite un acople sencillo, con componentes desmontables y fácilmente reemplazables. La estructura ligera permite el paso del aire lo cual asegura una adecuada ventilación de los componentes electrónicos.

Como desventaja, es necesario reconocer que la exposición de los componentes electrónicos al medio ambiente puede resultar en daño de los mismos o en fallas del ADEO debido a suciedad, humedad, etc.

En cuanto al cableado, la correcta disposición de este a lo largo de la estructura asegura una comunicación de datos fluida y asegura la alimentación de todos los dispositivos. En caso algún cable interfiriese con el recorrido de las hélices, este sería inmediatamente cortado por estas, ocasionando la falla total o parcial del SDEO.

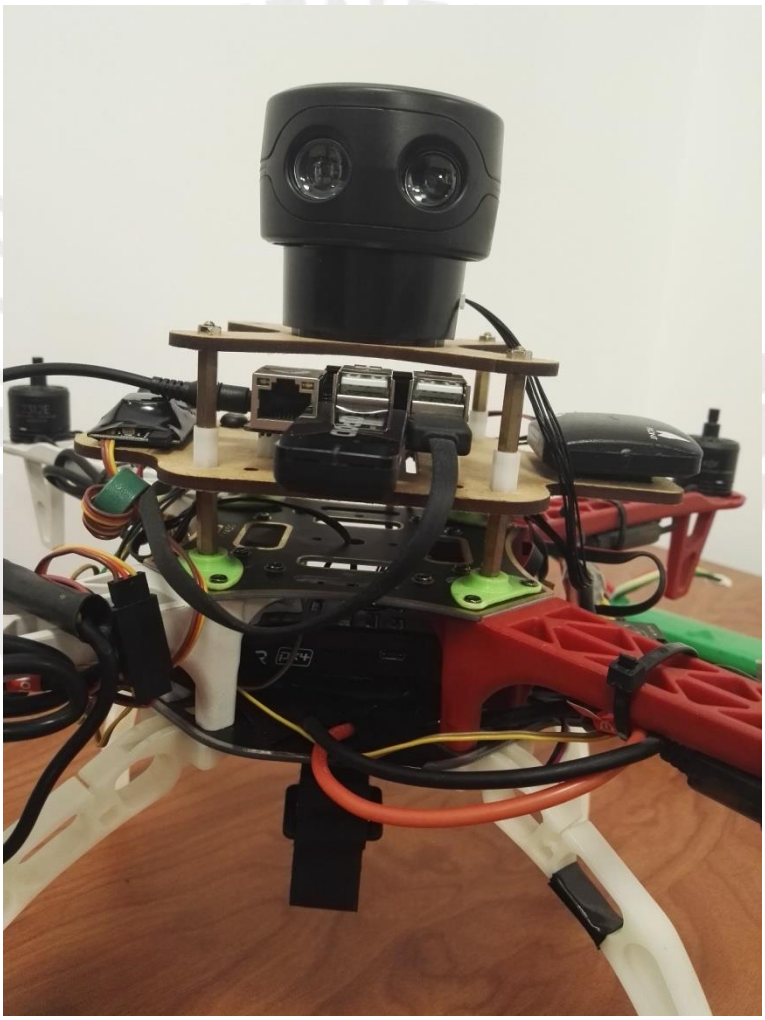

Figura 8. Estructura del SDEO con los componentes montados (Fuente: Imagen propia)

CAPÍTULO 3

DESARROLLO DEL SOFTWARE DEL SISTEMA DE DETECCIÓN Y EVASIÓN DE OBSTÁCULOS

En este capítulo de desarrollará el algoritmo de evasión de obstáculos teniendo como datos de entrada la información recopilada por el sensor Sweep LiDAR 360°. La información mencionada será procesada, ordenada y filtrada antes de ser utilizada por el algoritmo.

Se detallarán cada uno de los modos de vuelo desde la concepción de la idea hasta la lógica del algoritmo presente en cada uno de los modos de vuelo mediante gráficos. Además, se evidenciará la lógica de cada modo de vuelo mediante diagramas de flujo los cuales permitirán explicar cómo es llevado a cabo el procesamiento de la información y la toma de decisiones.

3.1 Consideraciones iniciales

Antes de detallar cada uno de los modos de vuelo, es pertinente mencionar ciertas consideraciones a tomar en cuenta para el desarrollo del SDEO.

Primero, el objetivo primordial del SANT es llegar a la meta. Es decir, trasladarse de un punto inicial hacia un punto destino o final. Para lograr dicho

objetivo de forma autónoma, es imprescindible que el controlador de vuelo Pixhawk se encuentre en modo de vuelo guiado.
Segundo, se define la fórmula de Haversine, necesaria para definir la meta y como llegar a esta.

Tercero, se explica el concepto de queue, para lograr multiprocesamiento.

Cuarto, existe una discordancia entre la orientación del sensor LiDAR y la orientación del Pixhawk, se ahondará en dicha diferencia.

Finalmente, se definen los vectores a utilizar en el algoritmo, se asignan estados a cada modo de vuelo y se brindan los parámetros a ingresar remotamente en el terminal de la computadora acompañante. Esto se realiza mediante un comando que permita la ejecución del ADEO.

Modo de vuelo "guiado":
La importancia de este modo de vuelo radica en que puede aceptar comandos por medio del protocolo de comunicación MAVLink. La meta se define por medio de coordenadas y la altura deseada en metros. Dichos datos son definidos en el ADEO antes de iniciar el vuelo. [25]

Fórmula de Haversine:
Permite calcular la distancia más corta posible entre 2 puntos sobre la superficie de la tierra. Esta fórmula toma en cuenta que la tierra no es una esfera, sino un geoide cuyo radio es 6,378 km, los ángulos utilizados se encuentran en radianes. La fórmula de Haversine permite un cálculo de distancia preciso. [26]

Queue:
El capítulo 8.10 de la librería estándar de Python define un queue como un módulo que implementa procesamiento en paralelo. Esto permite intercambio de información entre varias líneas de procesamiento. Realiza la función de un buffer, esto se evidencia en los 3 tipos de queue: FIFO (Primero en entrar,

primero en salir), LIFO (último en entrar, primero en salir), y Prioritario (se ordenan las entradas y el menor valor es el primero en salir). [27]

Orientación:

El sensor Sweep LiDAR 360° posee una orientación distinta al Pixhawk. De esta manera, al existir incompatibilidad, es necesaria una función la cual convierta vectores de LiDAR en vectores del vehículo (Pixhawk). Dicha función es definida como L2Veh en el ADEO. La diferencia en cuanto a la orientación se encuentra detallada en [23] y [28]. Además, puede observarse gráficamente en la Figura 9.

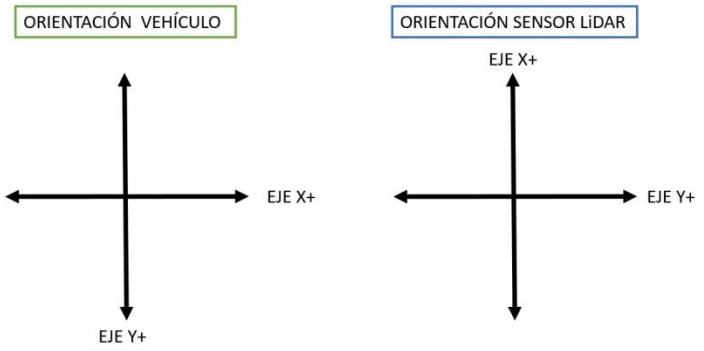

Figura 9. Diferencia entre la orientación del vehículo vs la orientación del LiDAR (Fuente: Elaboración Propia)

Vectores:

El ADEO hace uso de vectores para acceder a un modo de vuelo determinado. Dichos vectores se actualizan constantemente durante el vuelo variando su dimensión y orientación. En las figuras 10 y 11 pueden apreciarse los vectores mencionados a continuación.

- GTGVect: Vector que apunta hacia las coordenadas de la meta.
- AOVect: Vector que apunta hacia la dirección opuesta del objeto más cercano.
- VecTWall: Vector tangente a la pared.
- VecPWall: Vector perpendicular a la pared.
- ClosePVec: Vector que apunta al punto más cercano para el cálculo de VecPWall.

- FWVect: Vector resultante de la proporción entre VecTWall y VecPWall a través de αw y βw.
- MovVect: Vector que determina la velocidad del SANT por 1 segundo.

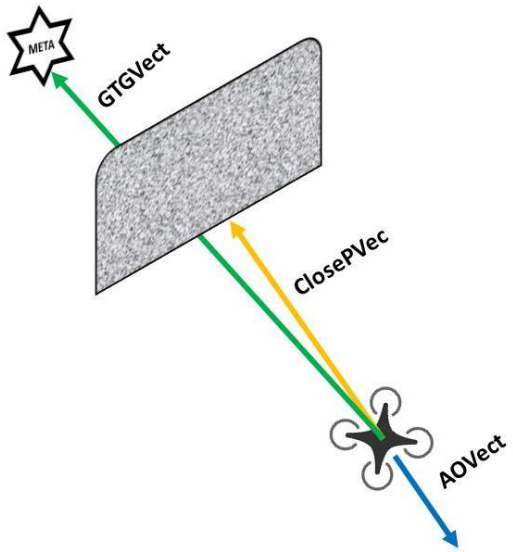

Figura 10. Vectores AOVect, ClosePVec y GTGVect (Fuente: Elaboración propia)

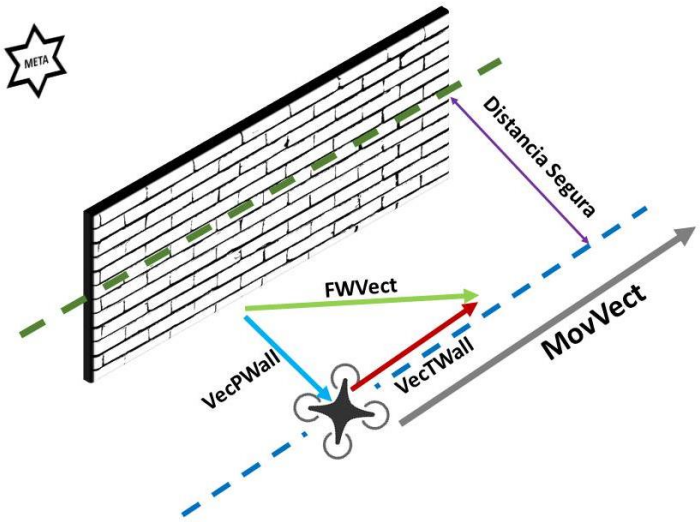

Figura 11. Vectores FWVect, VecPWall, VecTWall y MovVect (Fuente: Elaboración propia)

Estados:

El ADEO posee cuatro estados los cuales están asignados a cada modo de vuelo del SANT.

- Go to goal: 0
- Go to goal & Avoid Obstacle: 1
- Avoid Obstacle: 2
- Follow Wall: 3

Ejecución del algoritmo:

En cuanto a la ejecución del código, es necesario ingresar información preliminar. Esta información es:

- El puerto de conexión al controlador de vuelo Pixhawk.
- El puerto de conexión al sensor LiDAR.
- La velocidad de giro del motor.
- El radio de muestreo del LiDAR.
- El máximo número de puntos para reconstruir una pared.

Durante el vuelo, se imprimirán constantemente los siguientes valores en el terminal de la computadora en tierra:

- El modo de vuelo.
- La distancia a la meta.
- La dirección actual seguida.
- La distancia al punto más cercano.

3.2 Modos de vuelo

En cuanto a los modos de vuelo, el ADEO se basa en cuatro modos de vuelo. Cada modo de vuelo está asociado a un estado el cual se detalla en el apartado anterior. De esta manera, se ahondará en cada modo de vuelo

iniciando por el desarrollo del concepto hasta el desarrollo del diagrama de flujo.

Para comprender adecuadamente los modos de vuelo, se aprecia en la tabla 6 un resumen de los modos de vuelo existentes, los estados asignados a estos, el estado inicial requerido y por último las condiciones que deben o no deben cumplirse para acceder a estos.

Tabla 6. Resumen de los modos de vuelo (Fuente: Elaboración propia)

Modos de vuelo	Estado	Estado inicial requerido	Condiciones requeridas
Go to goal	0	1	**Si** se debe cumplir que: Mínima distancia al obstáculo mayor a cinco metros
Go to goal & Avoid Obstacle	1	0	**Si** se debe cumplir que: Mínima distancia al obstáculo menor o igual a cinco metros y mayor a dos metros
		2	**Si** se debe cumplir que: Mínima distancia al obstáculo mayor a dos metros
			No se debe cumplir que: (Mínima distancia al obstáculo menor a tres metros con veinticinco centímetros) y (el ángulo entre los vectores GTGVect y AOVect sea mayor a ciento cuarenta grados)
Avoid Obstacle	2	0	**No** se debe cumplir que: Mínima distancia al obstáculo sea menor o igual a cinco metros y mayor a dos metros
			Si se debe cumplir que: Mínima distancia al obstáculo menor o igual a dos metros
		1	**Si** se debe cumplir: Mínima distancia al obstáculo menor a dos metros
Follow Wall	3	1	**No** se debe cumplir: Mínima distancia al obstáculo menor a dos metros
			Si se debe cumplir que: (Mínima distancia al obstáculo menor a tres metros con veinticinco centímetros) y (el ángulo entre los vectores GTGVect y AOVect sea mayor a ciento cuarenta grados)
		2	**Si** se debe cumplir que: Mínima distancia al obstáculo mayor a dos metros
			Si se debe cumplir que: (Mínima distancia al obstáculo menor a tres metros con veinticinco centímetros) y (el ángulo entre los vectores GTGVect y AOVect sea mayor a ciento cuarenta grados)

3.2.1 “Go to goal” (Estado 0)

Este modo de vuelo tiene como finalidad conducir al SANT hacia la meta cuando las condiciones del entorno son ideales, operación normal del SANT. Es decir, no se presentan obstáculos entre el SANT y la meta.

Como se observó en el apartado 3.1, el modo de vuelo Go to goal tiene asignado el estado 0. El flujo seguido para acceder a dicho estado puede observarse en la figura 12.

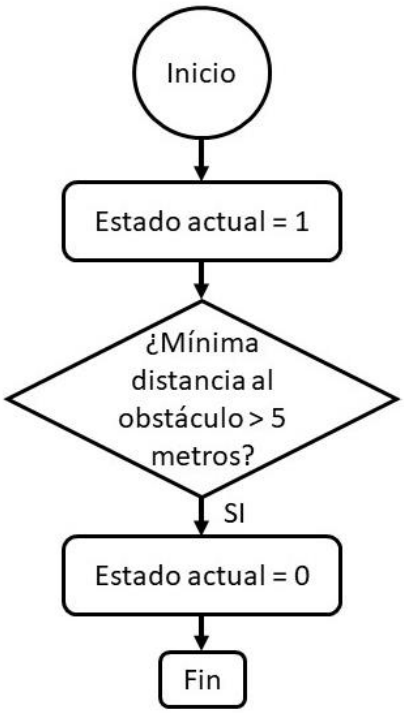

Figura 12. Diagrama de flujo del modo de vuelo Go to goal (Fuente: Elaboración propia)

3.2.2 “Go to goal & Avoid Obstacle” (Estado 1)

En este modo de vuelo el SANT se dirige hacia la meta con la tendencia de evadir el obstáculo. Como se observa en la tabla 6, se puede acceder a este modo de vuelo si la condición inicial es 0 o 2. La figura 13 muestra la interacción del SANT con el obstáculo y las distancias límites que enmarcan dicho modo de vuelo. El flujo correspondiente a dichos casos puede observarse en las figuras 14 y 15 respectivamente.

El valor de 3.25 metros corresponde a la distancia que se debe mantener en presencia de una pared, esto debido a que se desea diferenciar entre la necesidad de acceder al estado 1 o al estado 3. Además, el ángulo de 140° tiene justificación en el anexo 1.

En general, cuando el SANT se encuentre en el estado 1, el vector de movimiento se define por:
MovVect = VelSANT * (GTGF*GTGVect + AOF*AOVect)

Donde “VelSANT” es la velocidad del SANT y GTGVect y AOVect han sido definidos en el apartado 3.1.

GTGF = 0.8 (Factor de presencia del vector GTGVect en el estado 1).

AOF = 0.3 (Factor de presencia del vector AOVect en el estado 1).

Los valores mostrados anteriormente fueron obtenidos durante la creación de vectores que interactuan con el entorno. Los factores representan el porcentaje de presencia del vector en el estado mencionado.

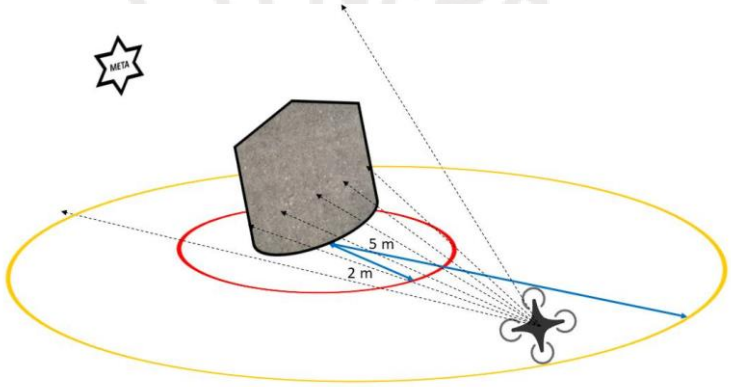

Figura 13. Representación gráfica del modo de vuelo Go to goal & Avoid Obstacle (Fuente: Elaboración propia)

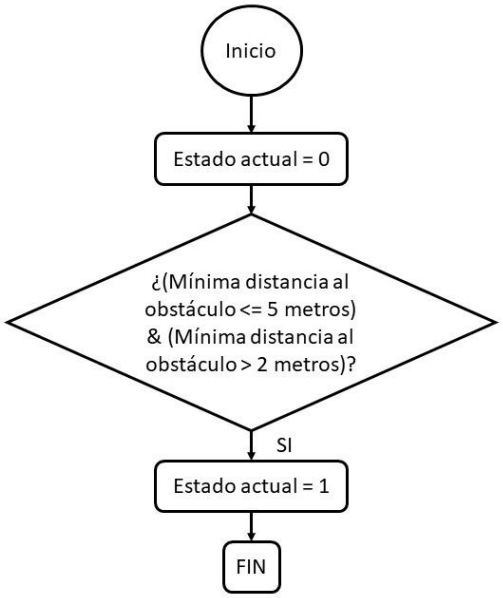

Figura 14. Diagrama de flujo del modo de vuelo Go to goal & Avoid Obstacle cuando el modo de vuelo anterior es Go to goal (Fuente: Elaboración propia)

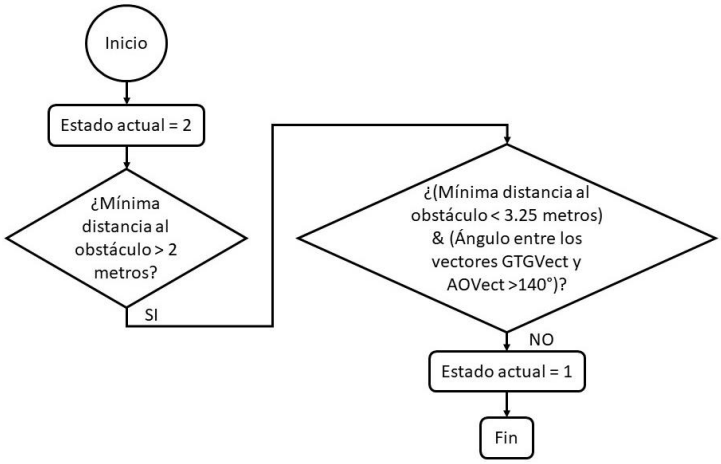

Figura 15. Diagrama de flujo del modo de vuelo Go to goal & Avoid Obstacle cuando el modo de vuelo anterior es Avoid Obstacle (Fuente: Elaboración propia)

3.2.3 “Avoid Obstacle” (Estado 2)

Este modo de vuelo representa la alternativa de emergencia para evitar una colisión directa. El ADEO elige este modo de vuelo cuando un obstáculo se interpone intempestivamente entre el SANT y la meta.

Como se observa en la tabla 6, se puede acceder a este modo de vuelo si la condición inicial es 0 o 1. La figura 16 muestra la interacción del SANT con el obstáculo y la distancia límite que define dicho modo de vuelo. El flujo correspondiente a dichos casos puede observarse en las figuras 17 y 18 respectivamente.

En general, cuando el SANT se encuentre en el estado 2, el vector de movimiento, MovVect, viene definido por:

MovVect = VelSANT * AOVect

Donde “VelSANT” es la velocidad del SANT y AOVect ha sido definido en el apartado 3.1.

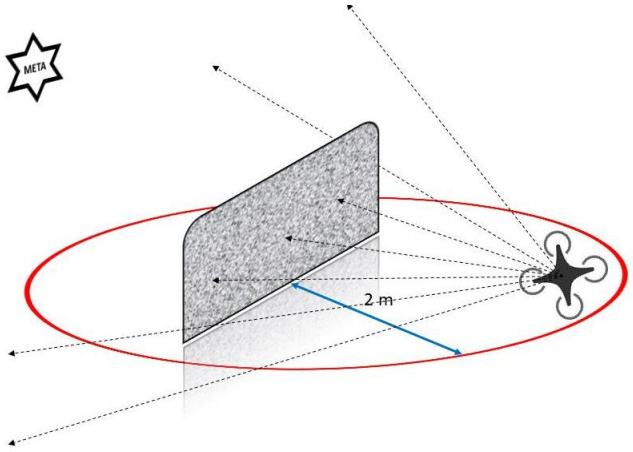

Figura 16. Representación gráfica del modo de vuelo Avoid Obstacle (Fuente: Elaboración propia)

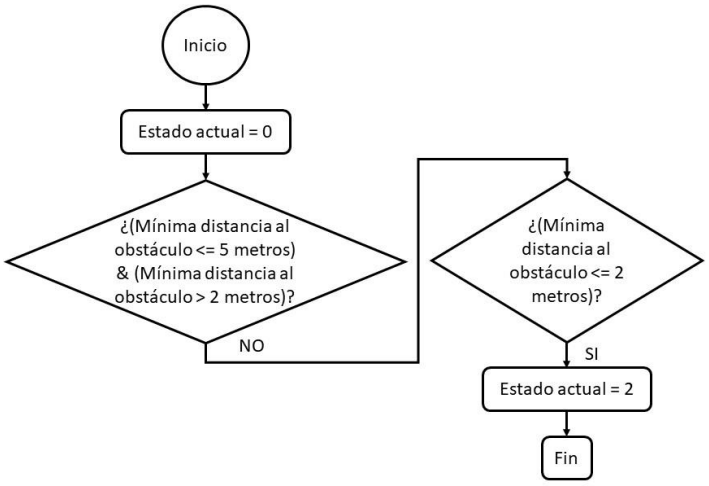

Figura 17. Diagrama de flujo del modo de vuelo Avoid Obstacle cuando el modo de vuelo anterior es Go to goal (Fuente: Elaboración propia)

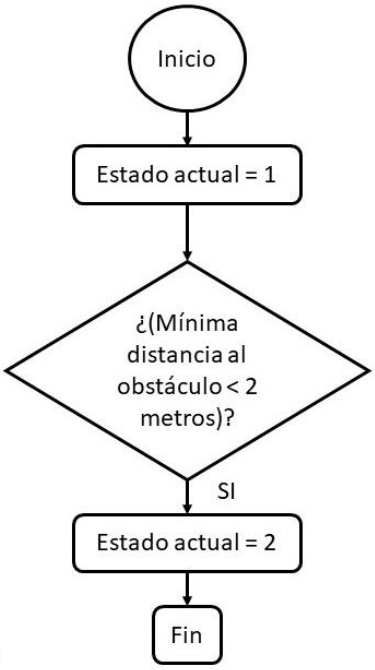

Figura 18. Diagrama de flujo del modo de vuelo Avoid Obstacle cuando el modo de vuelo anterior es Go to goal & Avoid Obstacle (Fuente: Elaboración propia)

3.2.4 “Follow Wall” (Estado 3)

Este modo de vuelo, el SANT detecta una continuidad en el obstáculo y este es asociado a una pared. Para poder alterar la trayectoria inicial, el SANT mantiene una distancia de seguridad previamente definida en el ADEO. De esta manera, se desplaza en una línea paralela a la recta asociada al obstáculo. Una de las fuentes utilizadas para elaborar dicho modo de vuelo puede encontrarse en [29]. Mayor información respecto a la implementación de algoritmos de control puede encontrarse en [30] y [31].

Como se observa en la tabla 6, se puede acceder a este modo de vuelo si la condición inicial es 1 o 2. La figura 19 muestra la interacción del SANT con el obstáculo y la recta asociada a este en base a la continuidad de los puntos ubicados. El flujo correspondiente a dichos casos puede observarse en las figuras 20 y 21 respectivamente. Además, el flujo seguido para definir el vector vecTWall puede observarse en la figura 22, mayor detalle referente a dicho vector puede encontrarse en el anexo 2

Por último, cuando el SANT se encuentre en el estado 3, el vector de movimiento, MovVect, viene definido por:

MovVect = VelSANT*FWVect

Donde “VelSANT” es la velocidad del SANT y FWVect ha sido definido previamente.

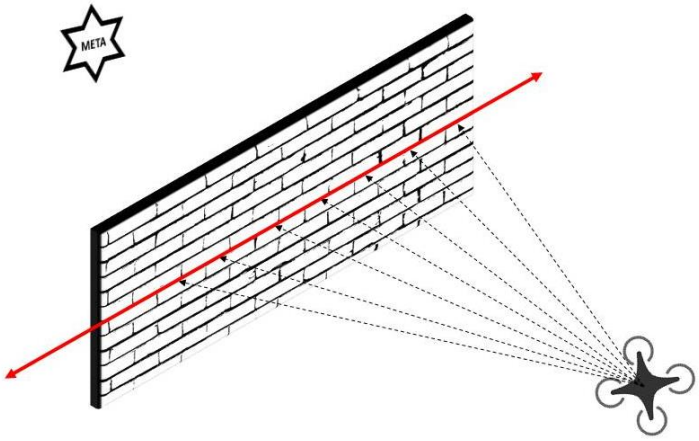

Figura 19. Representación gráfica del modo de vuelo Follow Wall (Fuente: Elaboración propia)

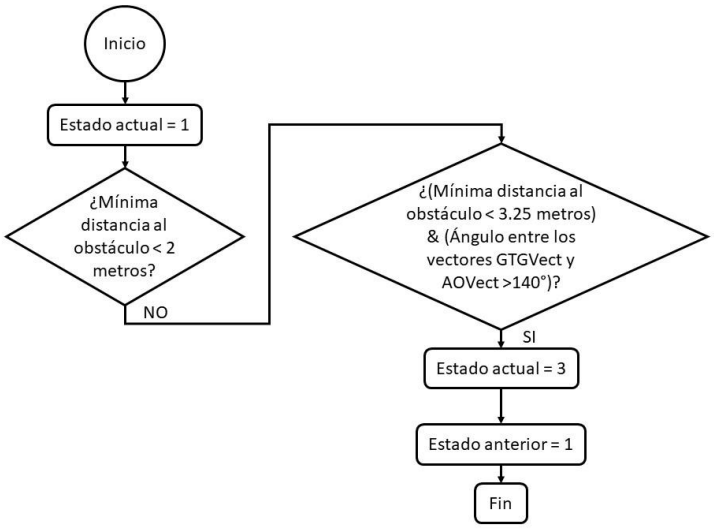

Figura 20. Diagrama de flujo del modo de vuelo Follow Wall cuando el modo de vuelo anterior es Go to goal & Avoid Obstacle (Fuente: Elaboración propia)

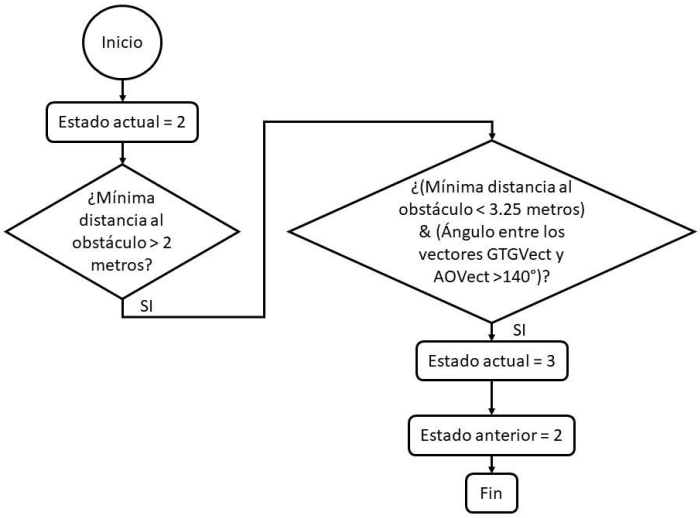

Figura 21. Diagrama de flujo del modo de vuelo Follow Wall cuando el modo de vuelo anterior es Avoid Obstacle (Fuente: Elaboración propia)

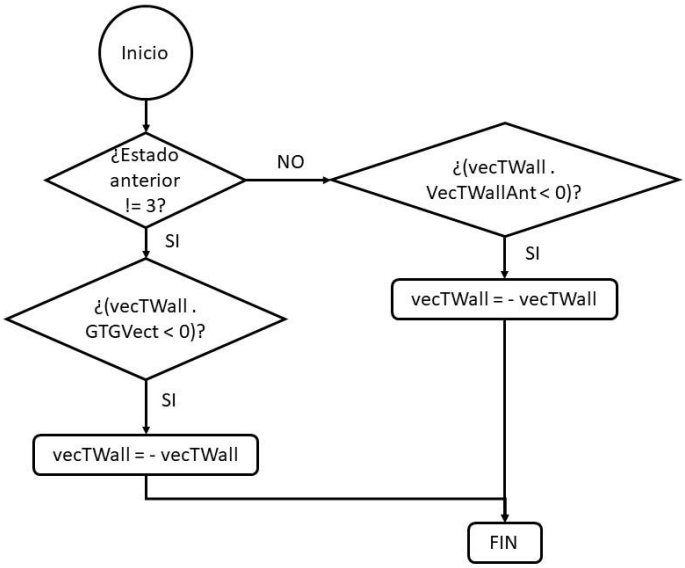

Figura 22. Diagrama de flujo para definir el vector vecTWall en base a condiciones iniciales. (Fuente: Elaboración propia)

CAPÍTULO 4

EJECUCIÓN DE PRUEBAS

En el presente capítulo se ejecutarán pruebas mediante simulación en Matlab y pruebas en campo. En cuanto a las pruebas simuladas, estas tienen como finalidad demostrar los modos de vuelo expuestos en el capítulo anterior, simular tres configuraciones de un biombo y mostrar de manera gráfica la trayectoria que sigue el SANT en base al vector de movimiento.

En cuanto a las pruebas en campo, se hará uso de un biombo elaborado con planchas de cartón de cartones de 1.60 m de alto por 2.4 m de ancho para simular un entorno de prueba. Dicho entorno de prueba se interpondrá entre el SANT y la meta. La finalidad de las pruebas en campo es corroborar el correcto funcionamiento del ADEO frente a estructuras planas.

4.1 Ejecución de pruebas simuladas mediante MATLAB

En este apartado se realizará la simulación de los cuatro modos de vuelo frente a un entorno en común. Se busca evidenciar el funcionamiento de cada modo de vuelo por separado. Además, se simulará la respuesta del ADEO frente a tres configuraciones de un biombo simulado.

4.1.1 Simulación de modos de vuelo

En este apartado se simularán los modos de vuelo empleados en el ADEO. Cada uno de los códigos será explicado brevemente y se mostrará una captura de la simulación correspondiente. Además, el código empleado para cada simulación se encontrará disponible en el anexo respectivo.

4.1.1.1 Modo de vuelo “Go to Goal” (Estado 0)

En el siguiente modo de vuelo se hará uso del programa GTG.m el cual ha sido desarrollado en MATLAB. El código referido a dicho programa puede encontrarse en el anexo 3.

La situación simulada es la ideal. Es decir, aquella en la cual el sensor LiDAR no detecta obstáculos que se interpongan entre el SANT y la meta. Puede observarse en la Figura 23, que el SANT parte de la esquina superior izquierda (-9,9), hacia la meta ubicada en la esquina inferior derecha (8,1).

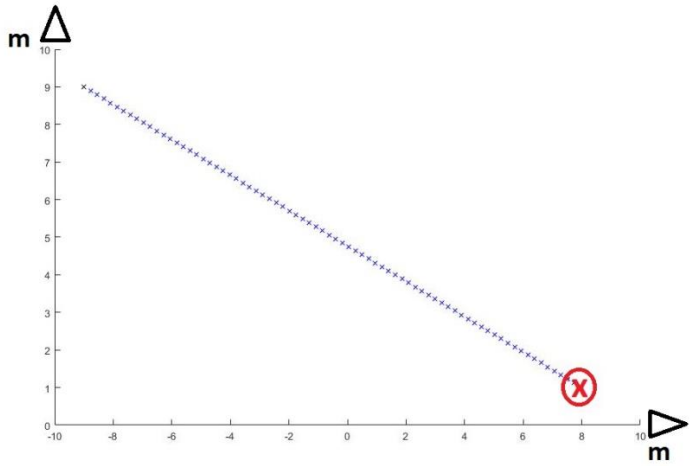

Figura 23. Simulación en MATLAB del modo de vuelo Go to Goal mediante el programa GTG.m (Fuente: Elaboración propia)

4.1.1.2 Modo de vuelo “Go to Goal & Avoid Obstacle” (Estado 1)

En el siguiente modo de vuelo se hará uso del programa GTGandAO.m el cual ha sido desarrollado en MATLAB. El código referido a dicho programa puede encontrarse en el anexo 4.

La situación simulada es aquella en la que el SANT se dirige hacia la meta y, un conjunto de obstáculos se interpone entre el SANT y la meta. Además, el SANT se encuentra a una distancia entre 2 y 5 metros del conjunto de obstáculos detectados. Frente a la situación descrita, el ADEO evade los obstáculos mientras intenta acercarse a la meta.

Puede observarse en la Figura 24, que el SANT parte de la posición (-2,2). En el camino hacia la meta el SANT se encuentra con el obstáculo representado por el cuadrado azul. El ADEO entra en estado 1 y bordea el obstáculo en sentido antihorario intentando no entrar en la zona de peligro delimitado por el círculo de color rojo. De esta manera el SANT alcanza la meta manteniendose entre 2 y 5 metros del obstáculo.

Es posible diferenciar 2 tipos de trazos en el recorrido del SANT representado mediante una sucesión de ‘x’ de color azul. El trazo de mayor grosor muestra la interacción del SANT con el obstáculo manteniendo el márgen del área de peligro (2 metros). El trazo de menor grosor representa la dirección del SANT orientandose hacia la meta.

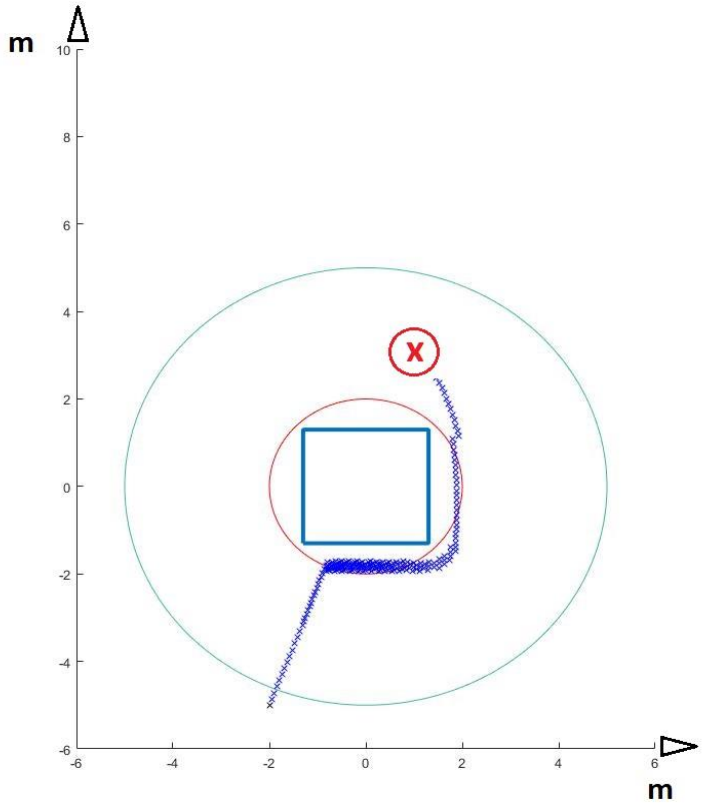

Figura 24. Simulación en MATLAB del modo de vuelo Go to Goal & Avoid Obstacle mediante el programa GTGandAO.m (Fuente: Elaboración propia)

4.1.1.3 Modo de vuelo “Avoid Obstacle” (Estado 2)

El siguiente modo de vuelo tiene como finalidad salvaguardar el SANT ante situaciones de emergencia. El ADEO ingresa a este modo de vuelo cuando el SANT se encuentra a menos de 2 metros del obstáculo. Dicho modo de vuelo permite que el SANT retroceda en dirección contraria al obstáculo con la finalidad de evitar una colisión.

Al simular dicho modo de vuelo únicamente se obtiene la respuesta de retroceso del SANT hasta salir de la zona de peligro. Una vez que ha salido de la zona de peligro vuelve a ingresar y el proceso se repite. De esta manera, el modo de vuelo Avoid Obstacle es únicamente de emergencia y representa un complemento tanto para las simulaciones como en una situación real.

4.1.1.4 Modo de vuelo “Follow Wall” (Estado 3)

En el siguiente modo de vuelo se hará uso del programa FW.m el cual ha sido desarrollado en MATLAB. El código referido a dicho programa puede encontrarse en el anexo 5

La finalidad de dicho programa es detectar que el objeto que se interpone entre el SANT y la meta representa una pared. Una vez detectada, se define una distancia de precaución para recorrerla manteniendo la distancia de precaución.

En la Figura 25, se observa que el SANT parte de la posición (-5,-5). Los obstáculos se ven representados por rectángulos de colores. El modo de vuelo descrito mantiene una trayectoria lo más parecida al contorno de los obstáculos. Sin embargo, se observa que, en las esquinas, este comportamiento se altera. Dicha alteración es producto de los vectores usados para reconstruir la pared los cuales pierden simetría y por ende la trayectoria se distorsiona.

Una vez que el ADEO detecta que no hay obstáculos que se opongan entre el SANT y la meta, se ingresa nuevamente al estado 0 y el SANT se dirige hacia la meta de manera directa.

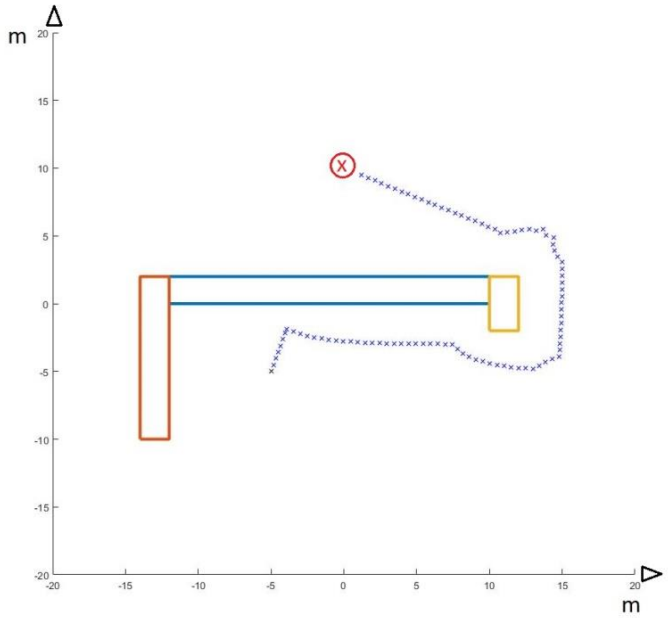

Figura 25. Simulación en MATLAB del modo de vuelo Follow Wall mediante el programa FW.m (Fuente: Elaboración propia)

4.1.2 Simulación de biombo

En el apartado anterior se simularon los modos de vuelo del ADEO. En este apartado se simularán 3 configuraciones de un biombo. Es importante resaltar que las configuraciones elegidas tienen la finalidad de evidenciar los modos de vuelo anteriormente descritos. De la misma manera, se realizarán pruebas con un biombo real las cuales serán descritas en el siguiente apartado.

4.1.2.1 Biombo extendido

La siguiente prueba tiene como finalidad simular una pared la cual se interpone entre el SANT y la meta. En la figura 26 se puede apreciar que el SANT mantiene una distancia de seguridad respecto al biombo. De esta manera, se está haciendo uso del modo de vuelo Follow Wall.

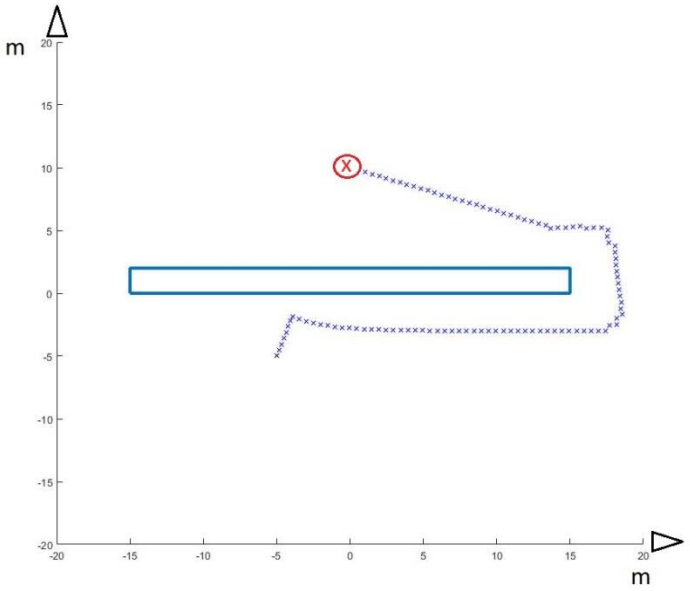

Figura 26. Simulación en MATLAB de un biombo extendido (Fuente: Elaboración propia)

4.1.2.2 Biombo escalonado

La siguiente prueba tiene como finalidad simular un obstáculo en desnivel el cual se interpone entre el SANT y la meta. En la figura 27 se puede apreciar que el SANT ingresa en el primer espacio escalonado, pero lograr salir al detectar el espacio reducido y la imposibilidad de alcanzar la meta. Al acercarse al segundo espacio escalonado, el SANT posee la tendencia a entrar al espacio escalonado, sin embargo, no ingresa debido a la posición del vector GTGVect. De esta manera, se está haciendo uso del modo de vuelo Go to goal & Avoid Obstacle en complemento con el modo de vuelo Follow Wall.

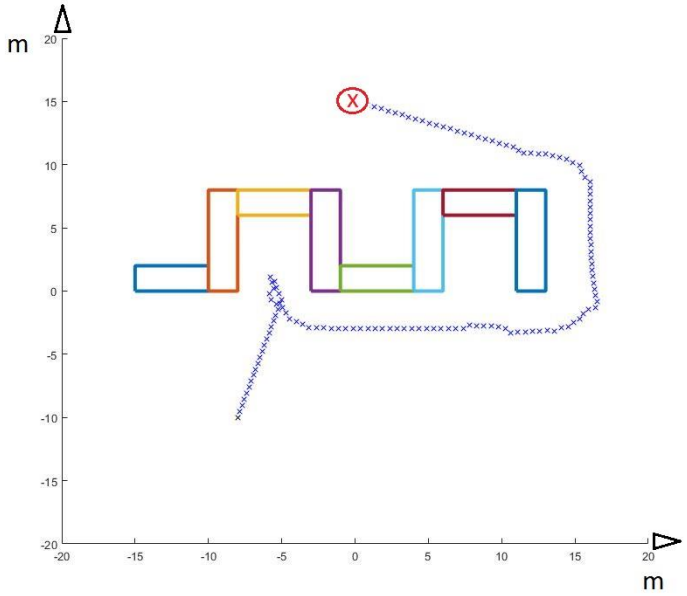

Figura 27. Simulación en MATLAB de un biombo escalonado (Fuente: Elaboración propia)

4.1.2.3 Biombo en “S”

La siguiente prueba tiene como finalidad simular un laberinto simple en forma de “S” para poder visualizar el comportamiento del ADEO en entornos con una única salida y entrada. Se puede observar en la figura 28 que el SANT se encuentra en el la parte inferior y la meta se encuentra ubicada en la parte superior.

El SANT parte de la posición (0,-10), se desplaza en sentido horario y al detectar las paredes cambia de rumbo en sentido contrario. A partir de este punto el SANT hace uso del modo Follow Wall y se desplaza a través del contorno del laberinto hasta ingresar a la parte superior. Se observa que en la esquina superior derecha el SANT hace uso del modo de vuelo Go to goal & Avoid Obstacle, para finalmente alcanzar la meta mediante el modo de vuelo Follow Wall.

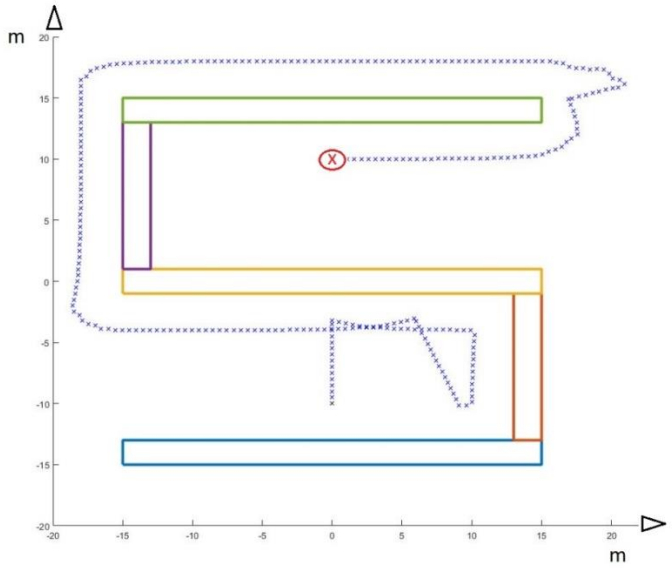

Figura 28. Simulación en MATLAB de un biombo en forma de "S" (Fuente: Elaboración propia)

4.2 Ejecución de pruebas reales

En la siguiente etapa se realizan las pruebas reales bajo un entorno controlado. Para la ejecución de dichas pruebas se requiere de un espacio amplio al aire libre el cual permita filmar las pruebas a una altura prudencial. Además, se requiere que el lugar elegido cuente con cobertura celular. Todo ello será posteriormente explicado en consideraciones iniciales. Las pruebas realizadas se llevaron a cabo mediante la ejecución del ADEO el cual se encuentra en el anexo 6.

4.2.1 Pruebas con biombo

Para la ejecución de las siguientes pruebas es necesario hacer uso de un biombo. El biombo utilizado consta de cuatro cajas de cartón las cuales al extenderse poseen una altura de 1.60 m de alto y 2.4 m de largo cada una. Finalmente, el biombo completo cuenta con una altura de 1.60 m de alto y 9.6 m de largo. Debido al material empleado, el biombo no puede sostenerse por sí mismo. De esta manera, es importante ubicar personas que sostengan el biombo al menos cada 2.5 m.

4.2.1.1 Consideraciones Iniciales

En cuanto a las consideraciones iniciales cabe mencionar que para realizar las pruebas de manera controlada se hizo uso de telemetría. De esta manera se colocó una antena de 915Mhz en el puerto de telemetría 1 del Pixhawk y otra en la laptop desde la cual se ejecutó el programa.

El mando de radiofrecuencia elegido fue el Futaba T8J. Dicho mando posee un selector de 3 niveles para poder seleccionar el modo de vuelo a utilizarse. Puede observarse en la figura 29 que se tienen 6 modos de vuelo a configurarse. Sin embargo, el selector tomará únicamente en cuenta los modos de vuelo 1, 4 y 6. Las demás opciones son dejadas en Loiter.

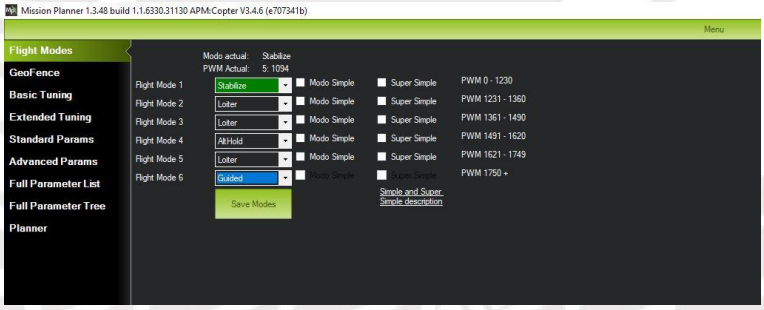

Figura 29. Configuración de modos de vuelo (Fuente: Elaboración propia)

De esta manera el primer modo de vuelo configurado es Stabilized, el segundo es Altitud Hold y finalmente el tercero es modo Guided. Al momento de ejecutar el ADEO, el SANT debe encontrarse en modo Guided. La configuración mencionada permite cambiar de modo de vuelo para evitar una colisión en caso el algoritmo no funcione adecuadamente y ponga en peligro la seguridad de las personas circundantes o al SANT.

Otra consideración importante es el acceso al sistema operativo de la computadora acompañante (Odroid C2) mediante el programa NoMachine. Dicho programa permite acceder remotamente al escritorio siempre y cuando este se encuentre en la misma red que la computadora de la cual se intenta acceder.

En este caso en particular, el programa fue instalado en la computadora acompañante y en la laptop de prueba. Se configuró una IP estática en la computadora acompañante para que se pueda acceder a esta siempre que se conecte a la red. Puede observarse en la figura 30 el programa configurado en la computadora acompañante. Dicho programa se ejecuta cada vez que el sistema inicia.

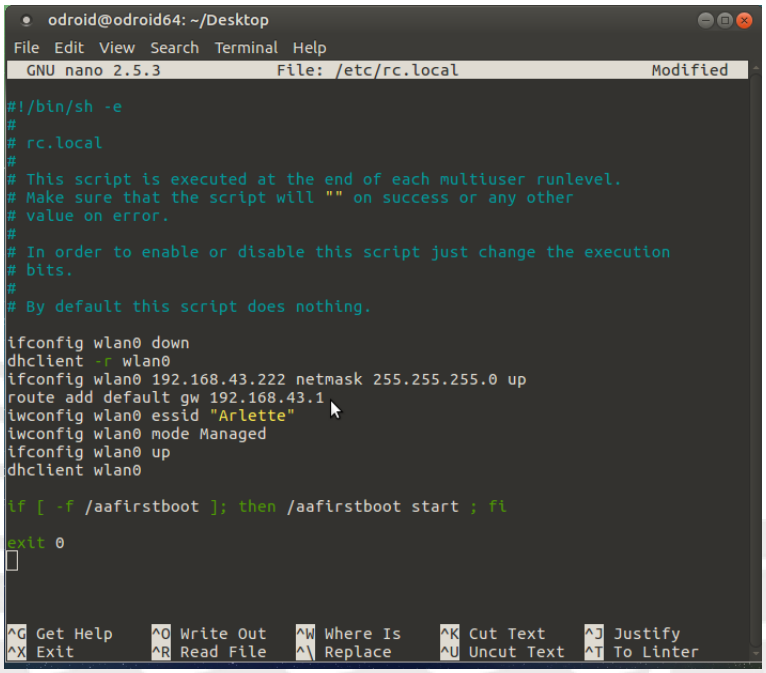

Figura 30. Configuración de IP Estática (Fuente: Elaboración propia)

Finalmente, el procedimiento para ejecutar el algoritmo es el siguiente:

1. Mantener el modo de vuelo en Guided haciendo uso del selector del control de radiofrecuencia.
2. Conectar la computadora acompañante y la laptop de prueba a una red de datos compartida desde un teléfono móvil con un SSID predefinido y sin contraseña.
3. Hacer uso del programa NoMachine para acceder remotamente al escritorio de la computadora acompañante.
4. Abrir una ventana de terminal y colocar: “cd”. Dicho código permite acceder a la carpeta Odroid donde se encuentra el ADEO.
5. Ejecutar el ADEO ingresando el siguiente código: Python [Algoritmo.py] [Conexión a Pixhawk] [Conexión a LiDAR] [Ratio de giro del motor] [Ratio de muestreo de LiDAR] [Máximo número de puntos necesarios para

reconstruir una pared]. Para este caso en particular se ingresa de la siguiente manera: python ADEO.py /dev/ttyUSB0 /dev/ttyUSB1 4 500 7. Esto puede apreciarse en la figura 31.

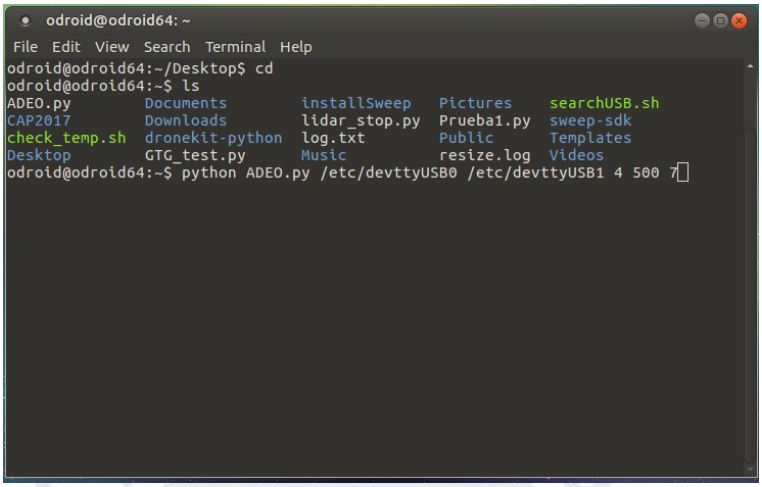

Figura 31. Ejecución del ADEO en el terminal de la computadora acompañante (Fuente: Elaboración propia)

4.2.1.2 Biombo extendido

La siguiente prueba tiene como finalidad evaluar el comportamiento del ADEO frente a un biombo el cual simula una pared. Finalmente contrastar dichos datos con los obtenidos mediante simulaciones del ADEO en Matlab. Además, se evalúa la reacción del ADEO frente a un obstáculo imprevisto. Para tal fin se hace uso de una pancarta.

La prueba con el biombo fue realizada 3 veces, 2 pruebas resultaron exitosas y 1 de ellas fallida. El lugar elegido para llevar a cabo las pruebas fue la cancha de minas de la PUCP. La prueba con la pancarta resultó exitosa y fue llevada a cabo en la cancha de Rugby de la PUCP.

Primera prueba exitosa:

En esta primera prueba se eligió como meta el punto ubicado en las coordenadas: (-12.072081, -77.081951). Se puede apreciar la meta elegida mediante el círculo rojo punteado.

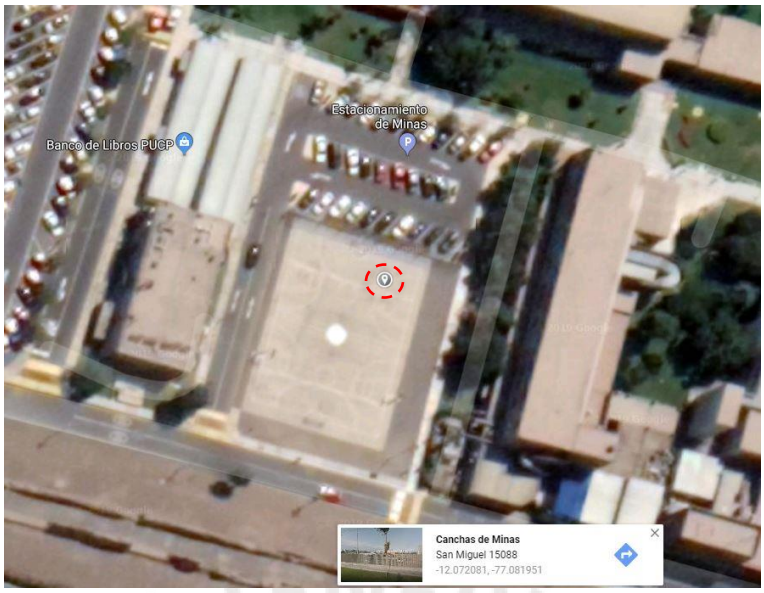

Figura 32. Coordenadas de la meta elegida para la primera prueba (Fuente: Google Maps)

En la figura 33 puede observarse el punto de partida el cual se ubica en el extremo opuesto a la meta elegida. Además, la flecha roja señaliza la dirección de vuelo tomada por el SANT.

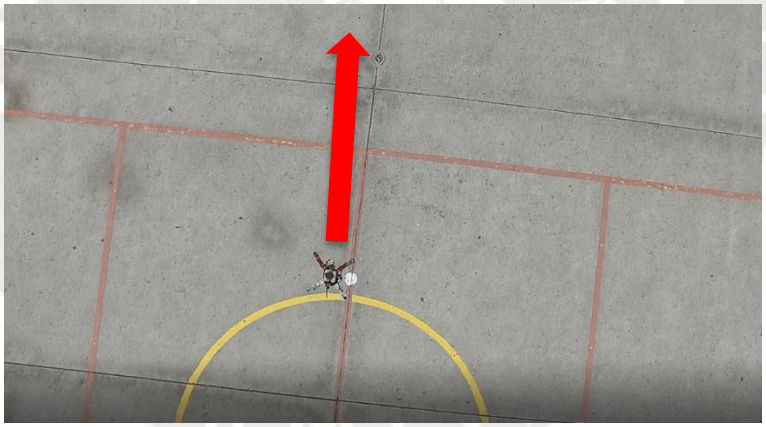

Figura 33. Punto de partida del SANT (Fuente: Elaboración propia)

En la figura 34 se puede apreciar el encuentro del SANT con el biombo mencionado en el apartado anterior. Se puede apreciar que se requiere de 4 personas para mantener el biombo en posición vertical. La flecha roja muestra la dirección tomada por el SANT en dicho instante.

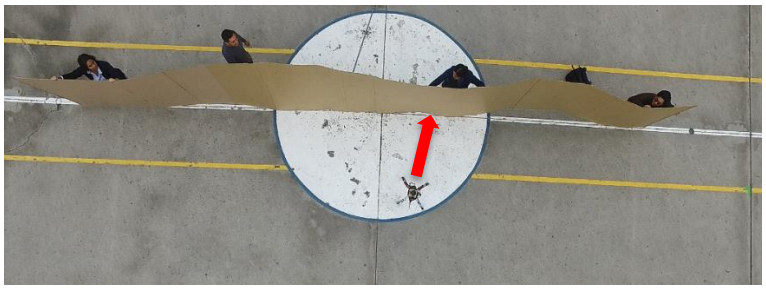

Figura 34. Interacción del SANT con el biombo (Fuente: Elaboración propia)

En la figura 35 se puede apreciar el primer intento del ADEO por evadir el biombo. En este caso en particular, la inestabilidad del biombo generó vértices en el lado derecho. De esta manera el ADEO optó por desplazarse hacia la izquierda. La flecha roja muestra el desplazamiento descrito.

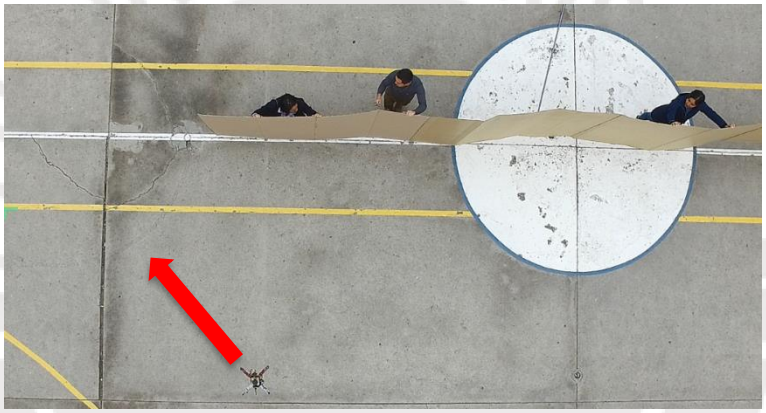

Figura 35. Primer intento de evasión (Fuente: Elaboración propia)

El SANT tiende a elevarse y descender verticalmente alrededor de la altura definida en el ADEO debido a que esta depende del barómetro en alturas menores a 10 m. Por encima de esta altura, el GPS es el que define la altura del SANT. En este caso en particular, la altura definida es de 1.3 m debido a la limitante impuesta por la altura del biombo. De esta manera, el LiDAR es capaz de detectar el biombo y el ADEO se podrá ejecutar adecuadamente.

Tomando en consideración la limitante del ADEO anteriormente expuesta, es necesario mencionar que el SANT tendía a elevarse por encima de la altura del biombo. En dichas ocasiones, el SANT se acercaba al biombo. Sin embargo, una vez que el SANT descendía nuevamente a la altura definida este detectaba el obstáculo en frente y retrocedía debido al modo de vuelo

Avoid Obstacle. La figura 36 muestra el momento en el cual el SANT se acerca al biombo debido a que el LiDAR no logra detectarlo.

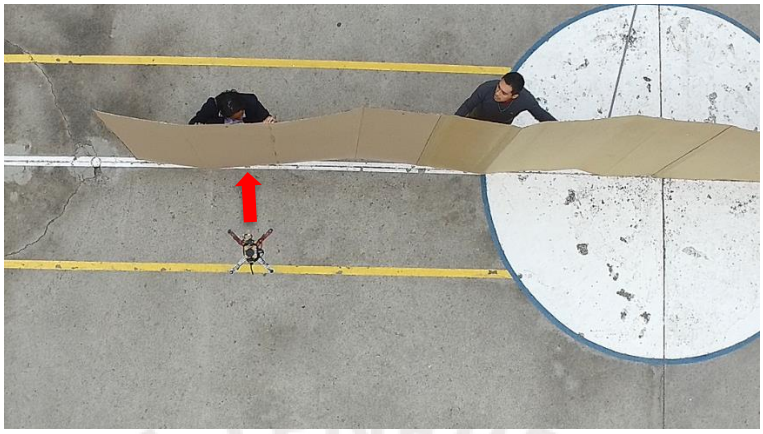

Figura 36. SANT por encima de la altura del biombo y acercamiento al biombo (Fuente: Elaboración propia)

En la figura 37 puede observarse el instante en el cual el ADEO desciende y el LiDAR es capaz de detectar nuevamente al biombo. Ante dicha situación, el ADEO ejecuta el modo de vuelo Avoid Obstacle para salir fuera del área de peligro.

Figura 37. Instante en el cual el SANT retrocede y la ruta es recalculada (Fuente: Elaboración propia)

En la figura 38 puede observarse que el ADEO optó por el lado derecho para evadir el biombo. Esto se debe a que el lado derecho se encuentra tensado y no presenta vértices los cuales ocasionan una lectura errónea. De esta manera el SANT se acerca de manera más eficiente hacia la meta. La flecha roja indica el desplazamiento durante el modo de vuelo Follow Wall.

Figura 38. Segundo intento de evasión (Fuente: Elaboración propia)

En la figura 39 puede observarse el momento en el cual el SANT evade adecuadamente el biombo. Este se desplaza por el lado derecho a una distancia prudencial del biombo. La flecha roja muestra el desplazamiento posterior a la evasión del biombo.

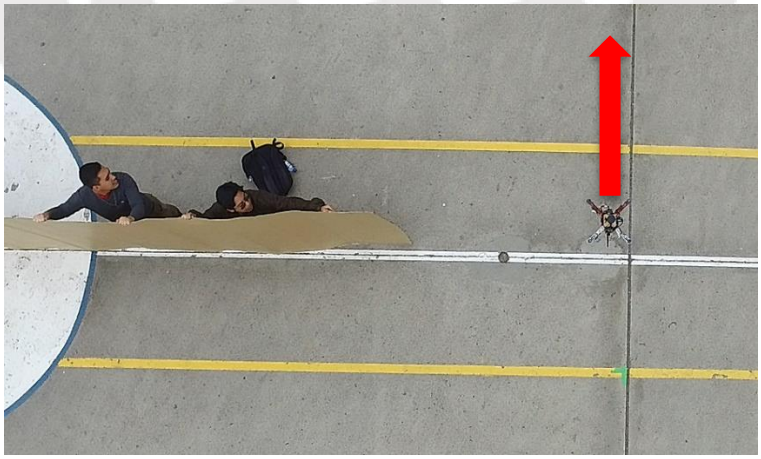

Figura 39. Evasión adecuada (Fuente: Elaboración propia)

En la figura 40 puede observarse el momento en el cual el SANT se posiciona adecuadamente en dirección a la meta elegida. La flecha roja muestra el posicionamiento del SANT para dirigirse a la meta.

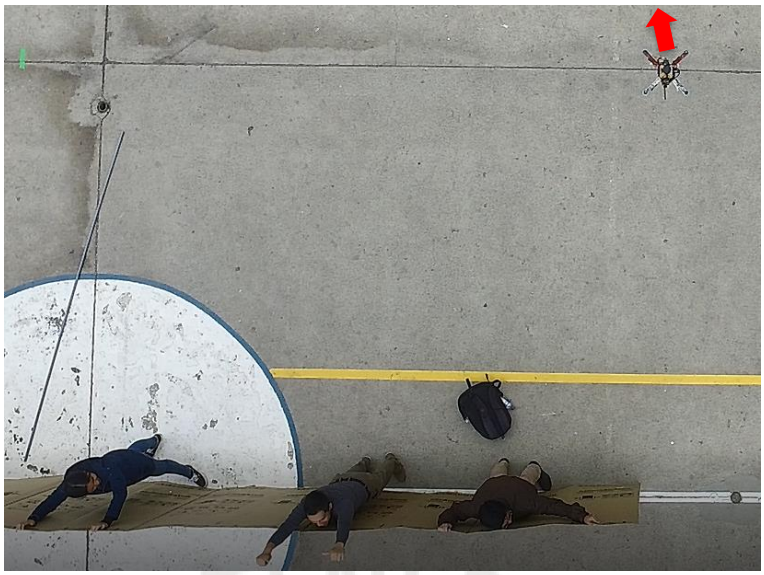

Figura 40. Evasión exitosa – Primera prueba (Fuente: Elaboración propia)

Segunda prueba exitosa:

En esta segunda prueba se eligió como meta el punto ubicado en las coordenadas: (-12.072094, -77.082056). Se puede apreciar la meta elegida mediante el círculo rojo punteado.

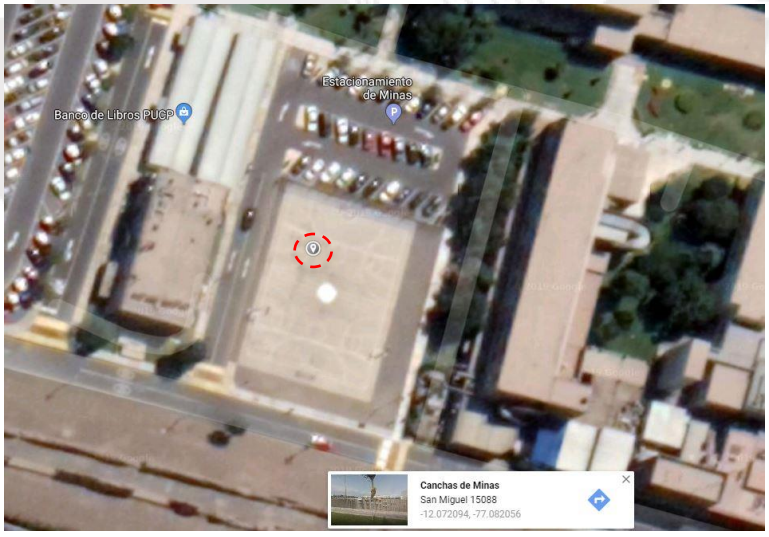

Figura 41. Coordenadas de la meta elegida para la segunda prueba (Fuente: Google Maps)

En la figura 42 puede observarse el punto de partida del SANT y la ubicación del biombo el cual se interpone entre el SANT y la meta elegida. La flecha roja indica la dirección tomada por el SANT al ejecutar el ADEO.

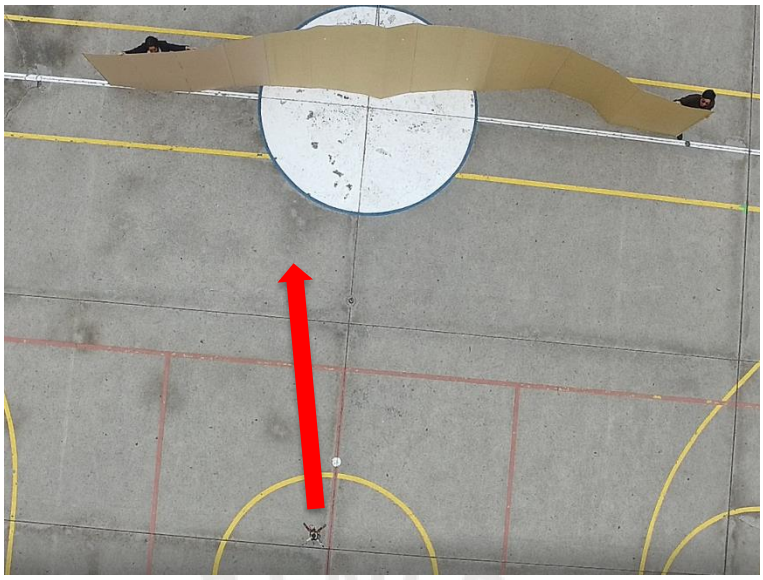

Figura 42. Encuentro del SANT frente al biombo (Fuente: Elaboración propia)

En la figura 43 puede observarse la dirección que toma el SANT para acercarse hacia la meta mientras evade el biombo. La flecha roja señaliza la dirección tomada por el SANT para evadir el biombo.

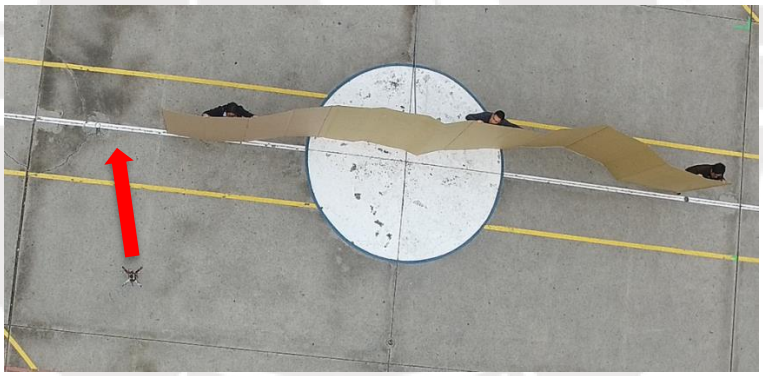

Figura 43. Evasión del biombo con dirección a la meta (Fuente: Elaboración propia)

En la figura 44 puede observarse el momento en el cual el SANT se posiciona adecuadamente con dirección a la meta elegida. La flecha roja muestra dicho posicionamiento.

Figura 44. Evasión exitosa – Segunda prueba (Fuente: Elaboración propia)

Tercera prueba exitosa:

En esta tercera prueba se eligió como meta el punto ubicado en las coordenadas: (-12.065232, -77.079766)

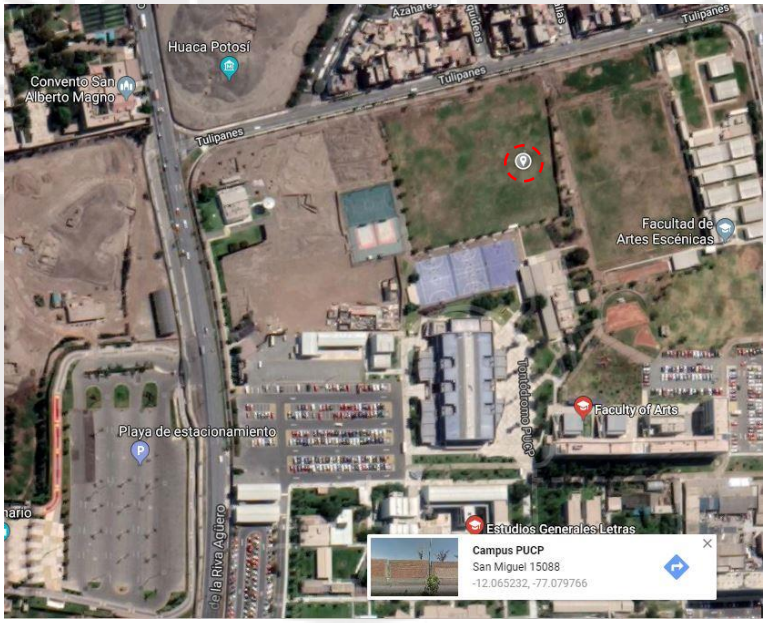

Figura 45. Coordenadas de la meta elegida para la tercera prueba (Fuente: Google Maps)

En la figura 46 puede observarse la interacción entre el SANT y el obstáculo. En este caso en particular, el obstáculo es una pancarta. La presente prueba pretende evidenciar el correcto funcionamiento del modo de vuelo Avoid Obstacle. Cabe mencionar que dicho modo de vuelo es el de emergencia.

La pancarta es levantada de manera imprevista frente al SANT. En dicho momento el SANT detecta el obstáculo presente y retrocede en dirección

contraria. Una vez que se ha alejado una distancia prudencial se desplaza hacia la derecha para evadirlo. Una vez se ha alejado de la zona de peligro, el SANT retoma su camino hacia la meta.

Figura 46. Prueba con pancarta (Fuente: Elaboración propia)

Prueba fallida:

En esta prueba, se mantuvo la meta de la primera prueba. Sin embargo, la inestabilidad del biombo generó vértices, se produjo una lectura del biombo la cual se alejaba de la posición real. Debido a ello, el SANT colisionó contra el biombo. Dicha prueba fallida representa una de las limitantes del ADEO el cual disminuye la eficiencia frente a estructuras inestables. En la figura 47 se observa el momento en el cual se produjo la colisión al intentar evadir el biombo con dirección a la meta.

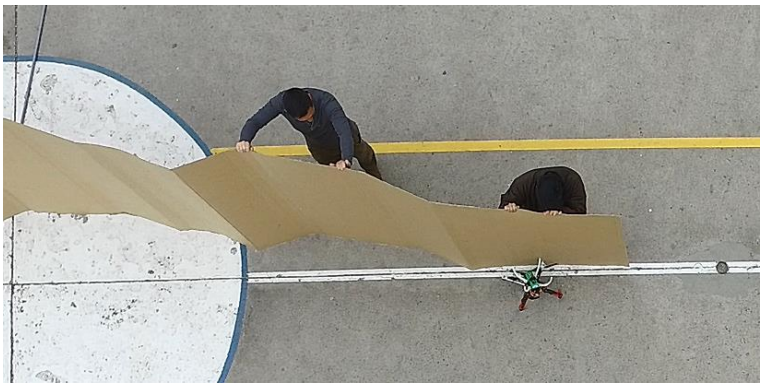

Figura 47. Evasión fallida - Colisión (Fuente: Elaboración propia)

En la figura 48 se puede apreciar la aproximación de la lectura obtenida del LiDAR en el instante de la colisión. Los círculos azules representan los puntos obtenidos por el sensor LiDAR respecto del biombo. Se puede observar como dichos puntos simulan el contorno del biombo visible en la figura 47. La recta verde corresponde a la lectura del ADEO frente a los datos presentes. La recta roja corresponde al contorno real del biombo ubicado frente al SANT. La flecha punteada de color naranja representa el desplazamiento del SANT con dirección a la meta en dicho instante.

Se observa la diferencia entre la recta detectada por el ADEO y la ubicación real del biombo. La diferencia observada en la figura 48 representa la causa principal de la colisión. De esta manera, una limitante del ADEO radica en la disminución de la eficiencia frente a estructuras no planas.

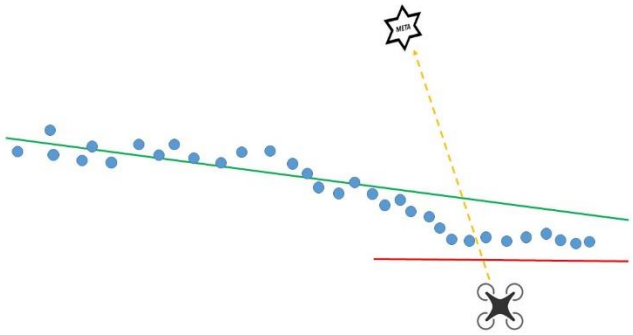

Figura 48. Aproximación de la lectura obtenida del LiDAR en el instante de la colisión (Fuente: Elaboración propia)

4.2.2 Pruebas con ventanas

Las siguientes pruebas tienen la finalidad de evaluar el desempeño del ADEO frente a ventanas. Para evitar daños a terceros o la propiedad privada, las siguientes pruebas se basarán únicamente en la lectura del LiDAR.

Se hará uso del programa Sweep Visualizer el cual permite visualizar en tiempo real los datos provenientes del sensor. Se contrastará la data obtenida del sensor frente a paredes con la data obtenida del sensor frente a ventanas. Dicha comparativa permitirá evaluar si la data que ingresa al ADEO es la adecuada para que este la procese y evite una colisión o, por el contrario, la data ingresada es insuficiente.

En la figura 49 se puede observar el sensor LiDAR elevado a la altura de las ventanas mediante un soporte. De esta manera es posible evaluar la señal obtenida de estas y compararlas con la obtenida de la pared contigua.

Figura 49. Soporte para pruebas en ventanas (Fuente: Elaboración propia)

En la figura 50 se puede observar la data obtenida del programa Sweep Visualizer. Se muestra gráficamente una carencia de puntos en orientación norte. Dicha orientación corresponde a la enfocada hacia las ventanas. En contraste, la cantidad de puntos obtenida en la parte izquierda muestra puntos continuos. Dichos puntos corresponden a la pared contigua.

En la figura 51 puede observarse los datos en formato tabla análogos a la figura 50. Aquellos puntos cuya intensidad de señal está por debajo de 128 serán descartados por el ADEO debido a que no representen una medida fiable para este.

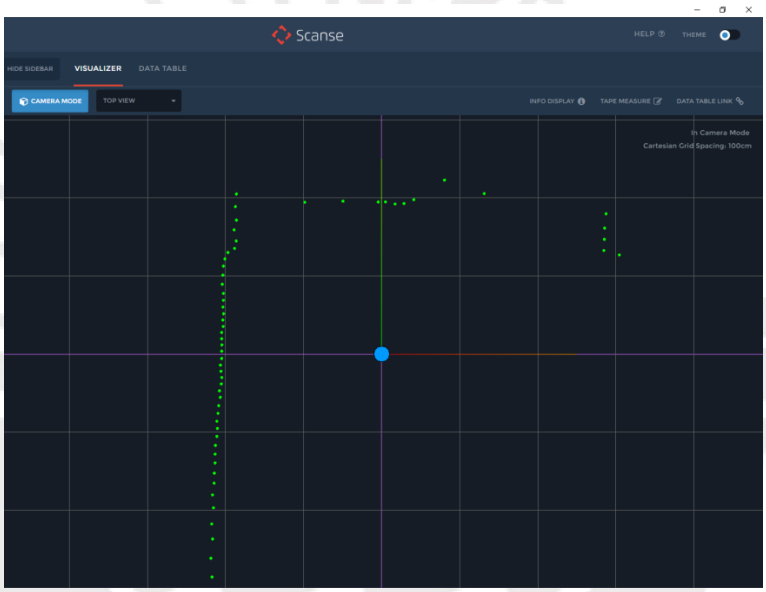

Figura 50. Gráfica obtenida del Sweep Visualizer (Fuente: Elaboración propia)

Puede observarse que los puntos obtenidos poseen intensidades de señal distintas entre sí. Aquellos puntos con intensidad de señal baja corresponden a la data obtenida de las ventanas, mientras que aquellos puntos que poseen una intensidad de señal elevada corresponden, en su mayoría, a la data obtenida de la pared contigua.

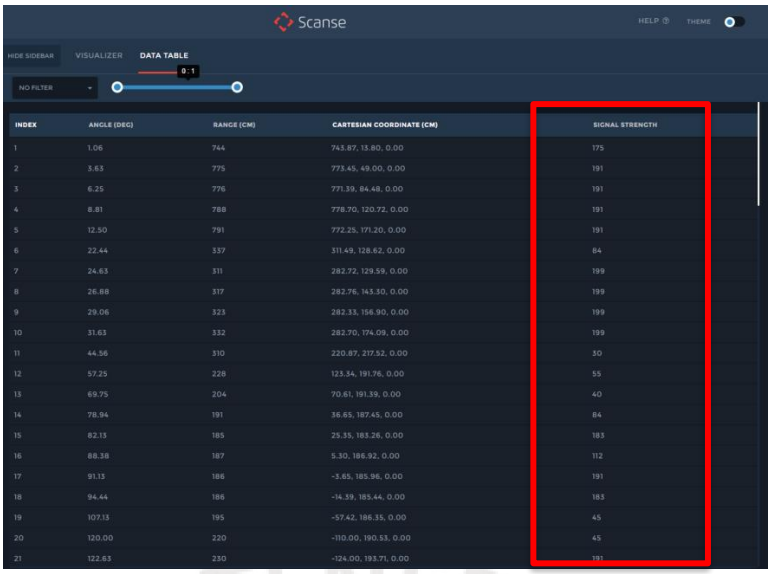

Figura 51. Tabla de datos obtenida del Sweep Visualizer (Fuente: Elaboración propia)

CONCLUSIONES

Las conclusiones de la presente tesis se encuentran ligadas directamente al objetivo general y los objetivos específicos planteados en el primer capítulo. Además, se brindarán conclusiones adicionales relacionadas al desarrollo del ADEO, las pruebas realizadas y al trabajo derivado. De esta manera, se concluye lo siguiente:

Respecto al Objetivo General:

- Se consiguió desarrollar un módulo electrónico capaz de evitar una colisión del SANT con obstáculos durante el vuelo en escenarios tales como bordear una estructura fija o evitar una colisión inminente con algún objeto que se interponga entre el SANT y la meta.

Respecto a los Objetivos Específicos:

- Se eligió el sensor Sweep LiDAR 360° el cual permite obtener información del entorno. Además, se seleccionó el Odroid C2 como la computadora acompañante con la finalidad procesar dicha información.
- Se diseñó una estructura compacta y ligera capaz de contener los dispositivos y permitir que el proceso de ensamblaje al SANT fuese sencillo.
- Se desarrolló un algoritmo de adaptación de rutas el cual permite a la computadora acompañante procesar la información procedente del sensor. Posteriormente, enviar el comando de vuelo hacia el Pixhawk a través del protocolo de comunicación Mavlink.
- Se llevaron a cabo 4 pruebas controladas mediante el uso de un biombo y una pancarta. Además, se realizaron pruebas en ventanas. Dichas pruebas permitieron demostrar el desempeño del módulo y evaluar las limitaciones del sistema desarrollado.

Conclusiones Adicionales:

Respecto a las pruebas realizadas con ventanas se evidencia, en la figura 51, un decremento en la intensidad de señal la cual se encuentra relacionada a la lectura procedente de las ventanas. Dicha información sería inmediatamente descartada por el ADEO. De esta manera se concluye que el ADEO no es capaz de detectar adecuadamente las ventanas y, por ende, el SANT colisionaría contra estas.

Respecto a la prueba fallida que culminó en una colisión, se concluye que una lectura de puntos disperso asociada a una estructura no plana o inestable puede generar un procesamiento inadecuado y conducir a una respuesta errónea por parte del ADEO. El modo de vuelo Avoid Obstacle, el cual había resultado exitoso en la prueba con la pancarta, no funcionó adecuadamente debido a lo mencionado anteriormente.

Respecto a la primera prueba, se concluye que la estabilidad vertical del biombo influye en gran medida en el desarrollo del ADEO. La ruta tomada inicialmente alejaba al SANT de la meta. Sin embargo, dicha ruta se tomó debido a la lectura obtenida del biombo en dicho instante.

Se concluye que el ADEO se comporta de manera similar a lo visto en las simulaciones. La diferencia radica en factores externos tales como la verticalidad y la estabilidad del obstáculo. Además, la intensidad del viento al momento de ejecutar las pruebas representa una perturbación.

Respecto a las limitaciones del SDEO, se concluye que la velocidad máxima de vuelo para que el ADEO procese adecuadamente la información y se ejecute adecuadamente es 0.5 m/s. Dicho valor fue obtenido en la prueba de la pancarta. Se probó la respuesta del ADEO a distintas velocidades y con una velocidad superior a la mencionada el ADEO posee una respuesta tardía lo cual conllevaría a una colisión.

Respecto a los trabajos derivados de la presente tesis, se desarrolló un paper el cual fue presentado en la conferencia internacional CASE 2018 llevada cabo en Múnich, Alemania. Además, dicho paper se encuentra publicado en la IEEE Xplore y es la referencia [40].

RECOMENDACIONES

En cuanto a las recomendaciones, es necesario mencionar que la presente tesis corresponde a un prototipo el cual puede ser mejorado hasta el punto de convertirse en un producto. Teniendo en consideración lo anteriormente expuesto, las recomendaciones en cuanto a hardware o software pretenden cubrir las falencias actuales y mejorar el desempeño del SDEO

En primer lugar, se recomienda implementar un altímetro. De esta manera el ADEO iniciará la trayectoria en un tiempo menor. Además, una vez iniciada la trayectoria, el SANT se mantendrá siempre a la altura elegida.

En segundo lugar, se recomienda implementar un log de vuela para que sea posible analizar adecuadamente cada vuelo realizado. En dicho log de vuelo debe registrarse el modo de vuelo elegido, la distancia al punto más cercano y la distancia a la meta en cada instante.

En tercer lugar, se recomienda fabricar una estructura envolvente la cual permita proteger al SDEO del exterior, salvaguardando los componentes electrónicos y previniendo el deterior de estos.

Finalmente, se recomienda ejecutar el programa Sweep Visualizer a la par del ADEO para conocer la información del entorno y contrastarla con la información obtenida del log de vuelo.

BIBLIOGRAFÍA

[1] TODRONE, «En 2017 se venderán 3 millones de drones | ToDrone», 2017. [En línea]. Disponible en: https://www.todrone.com/2017-venderan-3-millones-drones/. [Accedido: 25-sep-2018].

[2] BBVA, «¿Quién lidera el mercado de los drones?», 01-jul-2016. [En línea]. Disponible en: https://www.bbva.com/es/quien-lidera-mercado-drones/. [Accedido: 25-sep-2018].

[3] aereal, «aereal l Grabación aérea», 2018. [En línea]. Disponible en: https://www.aereal.pro/. [Accedido: 25-sep-2018].

[4] G. Wild y L. Handran, «Tech issues cause most drone accidents: research - RMIT University», 2016. [En línea]. Disponible en: https://www.rmit.edu.au/news/newsroom/media-releases-and-expert-comments/2016/august/tech-issues-cause-most-drone-accidents--research. [Accedido: 25-sep-2018].

[5] M. R. Ramli y D. S. Kim, «Obstacle Detection System using AR Drone Quadcopter with Ultrasonic Sensor», pp. 0-1.

[6] N. Gupta, J. Singh Makkar, y P. Pandey, «Obstacle detection and collision avoidance using ultrasonic sensors for RC multirotors», 2015 Int. Conf. Signal Process. Commun., pp. 419-423, 2015.

[7] National Oceanic and Atmospheric Administration, «What is Lidar and what is it used for? | American Geosciences Institute», 2018. [En línea]. Disponible en: https://www.americangeosciences.org/critical-issues/faq/what-lidar-and-what-it-used. [Accedido: 25-sep-2018].

[8] C. Kownacki, «A concept of laser scanner designed to realize 3D obstacle avoidance for a fixed-wing UAV», Robotica, vol. 34, n.o 2, pp. 243-257, feb. 2016.

[9] S. Ramasamy, A. Gardi, J. Liu, y R. Sabatini, «A laser obstacle detection and avoidance system for manned and unmanned aircraft applications», 2015 Int. Conf. Unmanned Aircr. Syst. ICUAS 2015, pp. 526-533, 2015.

[10] N. Gageik, P. Benz, y S. Montenegro, «Obstacle detection and collision avoidance for a UAV with complementary low-cost sensors», IEEE Access, vol. 3, pp. 599-609, 2015.

[11] L. Adouane, «Orbital Obstacle Avoidance Algorithm for Reliable and On-Line Mobile Robot Navigation». [En línea]. Disponible en: http://lounisadouane.online.fr/__Publications/ADOUANE_RoboticaObstacleAvoidance.pdf. [Accedido: 12-oct-2018].

[12] C. Shang et al., «Micro aerial vehicle autonomous flight control in tunnel environment», 2017 9th Int. Conf. Model. Identif. Control, n.o Icmic, pp. 93-98, 2017.

[13] P. Raja, «Optimal path planning of mobile robots: A review», Int. J. Phys. Sci., vol. 7, n.o 9, 2012.

[14] H. Choset y P. Pignon, «Coverage Path Planning: The Boustrophedon Cellular Decomposition», F. Serv. Robot., pp. 203-209, 1998.

[15] E. U. Acar, H. Choset, y P. N. Atkar, «Complete sensor-based coverage with extended-range detectors: A hierarchical decomposition in terms of critical points and voronoi diagrams», en IEEE International Conference on Intelligent Robots and Systems, 2001, vol. 3, pp. 1305-1311.

[16] S. Choueiry, M. Owayjan, H. Diab, y R. Achkar, «Mobile Robot Path Planning Using Genetic Algorithm in a Static Environment», 2019 4th Int. Conf. Adv. Comput. Tools Eng. Appl. ACTEA 2019, n.o January, 2019.

[17] «Global path planning for autonomous robot navigation using hybrid metaheuristic GA-PSO algorithm - IEEE Conference Publication». [En línea]. Disponible en: https://ieeexplore.ieee.org/abstract/document/6060543. [Accedido: 07-mar-2020].

[18] L. G. Freire Bouillon, «Página 12 - Perfiles IDS - Sistemas Aéreos no Tripulados», 2009. [En línea]. Disponible en: https://www.infodefensa.com/publlicaciones/perfiles-uas/files/assets/basic-html/page12.html. [Accedido: 25-sep-2018].

[19] M. Guillén, «Tipos de drones aéreos - Drone Spain». [En línea]. Disponible en: http://dronespain.pro/tipos-de-drones-aereos/. [Accedido: 25-sep-2018].

[20] «BeginnersGuide/NonProgrammers - Python Wiki». [En línea]. Disponible en: https://wiki.python.org/moin/BeginnersGuide/NonProgrammers. [Accedido: 30-oct-2018].

[21] «Introduction · MAVLink Developer Guide». [En línea]. Disponible en: https://mavlink.io/en/. [Accedido: 30-oct-2018].

[22] «XML Basics - XML Files». [En línea]. Disponible en: https://www.xmlfiles.com/xml/. [Accedido: 30-oct-2018].

[23] Scanse, «User's Manual and Technical Specifications - Scanse», 2017. [En línea]. Disponible en: www.scanse.io. [Accedido: 10-oct-2018].

[24] «Communicating with ODroid via MAVLink — Dev documentation». [En línea]. Disponible en: http://ardupilot.org/dev/docs/odroid-via-mavlink.html. [Accedido: 09-nov-2018].

[25] «Guided Mode — Copter documentation». [En línea]. Disponible en: http://ardupilot.org/copter/docs/ac2_guidedmode.html. [Accedido: 09-nov-2018].

[26] «Calculate distance and bearing between two Latitude/Longitude points using haversine formula in JavaScript». [En línea]. Disponible en: https://www.movable-type.co.uk/scripts/latlong.html. [Accedido: 31-oct-2018].

[27] «8.10. Queue — A synchronized queue class — Python 2.7.15 documentation». [En línea]. Disponible en: https://docs.python.org/2/library/queue.html. [Accedido: 31-oct-2018].

[28] H. Niwa, «[TAOGLAS GP.1575.25.4.A.02 DATASHEET]», Development, vol. 134, n.o 4, pp. 635-646, 2007.

[29] «Communicating with ODroid via MAVLink — Dev documentation». [En línea]. Disponible en: http://ardupilot.org/dev/docs/odroid-via-mavlink.html. [Accedido: 09-nov-2018].

[30] Dr. Magnus Egerstedt, «Control of Mobile Robots | Coursera». [En línea]. Disponible en: https://www.coursera.org/learn/mobile-robot. [Accedido: 09-nov-2018].

[31] Georgia Tech, «Control of Mobile Robots-1.1 Control of Mobile Robots - YouTube», 2013. [En línea]. Disponible en: https://www.youtube.com/watch?v=aSwCMK96NOw&list=PLp8ijpvp8iCvFDYdcXqqYU5Ibl_aOqwjr. [Accedido: 09-nov-2018].

[32] S. Angulo, «Estados Unidos actualiza su regulación de drones • ENTER.CO», 2016. [En línea]. Disponible en: http://www.enter.co/especiales/experiencia-dron/estados-unidos-actualiza-su-regulacion-de-drones/. [Accedido: 25-sep-2018].

[33] Gobierno del Perú, :::«Ministerio de Transportes y Comunicaciones»: [En línea]. Disponible en: http://portal.mtc.gob.pe/transportes/aeronautica_civil/index.html. [Accedido: 25-sep-2018].

[34] C. Escura Forcada, «La electrónica de vuelo en un drone - VueloArtificial». [En línea]. Disponible en: https://vueloartificial.com/introduccion/primeros-pasos/la-electronica-de-vuelo/. [Accedido: 25-sep-2018].

[35] «Qué es un LIDAR, y cómo funciona el sensor más caro de los coches autónomos». [En línea]. Disponible en: https://www.motorpasion.com/tecnologia/que-es-un-lidar-y-como-funciona-el-sistema-de-medicion-y-deteccion-de-objetos-mediante-laser. [Accedido: 05-sep-2018].

[36] N. O. and A. A. US Department of Commerce, «What is LIDAR», 2018. [En línea]. Disponible en: https://oceanservice.noaa.gov/facts/lidar.html. [Accedido: 25-sep-2018].

[37] LiDAR - UK, «How does LiDAR work? The science behind the technology. », 2018. [En línea]. Disponible en: http://www.lidar-uk.com/how-lidar-works/. [Accedido: 25-sep-2018].

[38] S. J. Pittman, B. Costa, y L. M. Wedding, Coral Reef Remote Sensing, n.o May 2014. 2013.

[39] Humboldt State University, «Lidar Applications», 2015. [En línea]. Disponible en: http://gsp.humboldt.edu/olm_2015/Courses/GSP_216_Online/lesson7-1/applications.html. [Accedido: 25-sep-2018].

[40] J. Gonzalez, A. Chávez, J. Paredes, y C. Saito, «Obstacle Detection and Avoidance Device for Multirotor UAVs through interface with Pixhawk Flight Controller», IEEE Int. Conf. Autom. Sci. Eng., vol. 2018-August, pp. 110-115, 2018.

ANEXOS

ANEXO 1

La fórmula utilizada para definir el ángulo de 140° es:

$$\frac{\sum(GTGVect \times AOVect)}{\|GTGVect\| \times \|AOVect\|} < -0.75$$

El valor de -0.75 es el coseno de -138.59° el cual representa el ángulo entre los vectores: GTGVect y AOVect. En caso el ángulo entre estos dos vectores sea menor a 140°, el modo de vuelo estará orientado hacia Go to goal & Avoid Obstacle. En caso el ángulo sea mayor a 140°, el modo de vuelo estará orientado hacia Follow Wall.

El valor de 140° ha sido elegido debido a un factor de seguridad para diferenciar adecuadamente ambos modos de vuelo.

ANEXO 2

Se definen los siguientes vectores:

vecTWallAnt: Vector que guarda el valor anterior del vector tangente a la pared.

nvecPWall: vector de transformación de coordenadas del vector vecPWall mediante la función L2Veh. Dicha función es definida en el apartado 3.1.

$$vecTWall = \frac{vecTWall}{\|vecTWall\|}$$

Luego, se define el vector vecPWall de la siguiente manera:

$$vecPWall = closePVec - (closePVec \,.\, vecTWall) \times vecTWall$$

$$vecPWall = vecPWall - dkept \times (\frac{vecPWall}{\|vecPWall\|})$$

Además, es necesario calcular el vector FWVect de la siguiente manera:

$$FWVect = (\alpha W \times vecTWall) + (\beta W \times vecPWall)$$

(αW y βW son 0.8 y 0.3 respectivamente).

Una vez hallado el vector FWVect, se debe utilizar la función L2Veh para transformarlo a las coordenadas del vehículo:

$$FEVect = L2Veh \times (\frac{FWVect}{\|FWVect\|})$$

ANEXO 3

```
close all
clear all
clc

goal = [8.0,1.0]; %Meta final.
rad = 0.5; %Radio de meta. Dentro del mismo, se acaba la simulacion.
start = [-9.0,9.0]; %Lugar donde inicia la simulcion.

%Se dibuja la meta, el lugar de inicio y un circulo rodeando el area de
%meta.
figure(1)
hold on
scatter(goal(1),goal(2),[],'x','r')
circle(goal(1),goal(2),rad);
scatter(start(1),start(2),[],'x','k')
axis([-10 10 0 10])
hold off
drawnow
pause(1)

pos = start;
vecScale = 0.25; %Velocidad: que tanto deberia avanzar en cada paso.
kk = 1;%Variable que acaba el programa si es que toma demasiado tiempo para evitar
bucles infinitos.

while(1)

    vect = vecScale.*(goal-pos)/norm(goal-pos); %Multiplica vector unitario apuntando a la
meta por el paso elegido.
    pos = pos + vect;
    hold on
```

```
    scatter(pos(1),pos(2),[],'x','b')
    hold off
    drawnow
    pause(0.05)

    if norm(goal-pos)<=rad %Se verifica si el punto actual ha llegado al area de la meta.
      break;
    end

    kk = kk+1;

    if kk > 1000    %Si es que se llevan a cabo mas de 1000 iteraciones, el programa se
acaba.
      break;
    end

end
```

ANEXO 4

```
close all
clear all
clc

kk = 1; %Contador de Simulacion
pos = [-2,-5]; %Posicion Inicial
goal = [2,6]; %Meta
rad = 0.50;

ax = [-6 6 -6 10]; %Limites del dibujo (X e Y respectivamente) (CAMBIABLE)
xPosTxt = min(ax(1:2)) + 1; %Posicion x de texto
yPosTxt = max(ax(3:4)) - 1; %Posicion y de texto

GTGVect = [0 0]; %Vector que apunta hacia meta
AOVect = [0 0]; %Vector que evade obstaculos
movVect = [0 0]; %Vector de direccion de movimiento

rect1 = rectm(-1.3,1.3,-1.3,1.3); %Rectangulo de pared
rectMat = [rect1];

vel = 0.25; %Velocidad (magnitud de vector de direccion)

sect = 40; %Numero de sectores a evaluar
sectDeg = 360/40;
sensInf = zeros(sect,5); %Informacion de sensor LIDAR

maxDistAO = 6; %Distancia maxima tomada para obtener vector de obstacle avoidance
pointsAO = zeros(sect,2); %Vectores de cada punto de LIDAR identificado
AOFact = ones(1,sect); %Factores que dan importancia a ciertos valores de LIDAR
dependiendo de su orientacion

minDist = 0;
detTrue = 0;

colFlag = 0; %Utilizado para determinar colsion (Bandera de colisión)
h = zeros(1,3);

sText = '';
%Dibujando graficos iniciales
```

```
figure(1)
hold on
for i=1:length(rectMat)
  rectMat(i).dRect();
end

%rect1.dRect(); %Dibujando rectangulos
scatter(goal(1),goal(2),[],'x','r')%Dibujando meta
circle(goal(1),goal(2),rad);%Dibujando circulo de meta
scatter(pos(1),pos(2),[],'x','k') %Dibujando posicion inicial
axis([-6 6 -6 10])
C1=sirkl([0,0],2)
C2=sirkl([0,0],5)
hold off
drawnow
pause(1)

while(1)

   GTGVect = vel*((goal-pos)/norm(goal-pos));

   minDist = 0;
   detTrue = 0;

   for i = 1:sect
     %sensInf(i,:) =  rect1.wallDist(pos,sectDeg*(i-1))';
     [sensInf(i,1),sensInf(i,2),sensInf(i,3),sensInf(i,4),sensInf(i,5)] =
wallDistMult(rectMat,pos,sectDeg*(i-1)); %Recogiendo informacion de sensor en cada
angulo evaluado
     if sensInf(i,4)==0
        if detTrue == 0
           minDist = sensInf(i,3);
           detTrue = 1;
        else
           if minDist>sensInf(i,3)
              minDist = sensInf(i,3);
           end
        end
     end

     if sensInf(i,4)==2 %Si posicion se encuentra dentro de rectangulo, terminar programa
por colision
        colFlag = 1;
        disp('Colision')
        break;
     end
   end

   if colFlag == 1
      break;
   end

   for i = 1:sect %Se evalua cada sector

      if sensInf(i,4) == 1
         pointsAO(i,:) = AOFact(i).*maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))]; %Si no
se pudo leer punto en esa direccion, se asigna un vaolor arbitrario a ese vector.
      elseif sensInf(i,4) == 0
         if sensInf(i,3)<maxDistAO %Si se leyo un punto en esa direccion se asigna un
magnitud menor a la arbitraria.
```

```
            pointsAO(i,:) = sensInf(i,3).*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
        else
            pointsAO(i,:) = maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
        end
    end

end

AOVect = sum(pointsAO)/sect;
AOVect = vel*(AOVect/norm(AOVect));

if minDist<2 && detTrue == 1
    if minDist <= 0.5
        movVect = AOVect;
    else
        movVect = 0.8*GTGVect + 0.2*AOVect;
    end
else
   movVect = GTGVect;
end

pos = pos + movVect; %Se actualiza la posicion basada en el vector de movimiento.

hold on
scatter(pos(1),pos(2),[],'x','b') %Se dibuja el nuevo punto.
h(1) = quiver(pos(1)-movVect(1),pos(2)-
movVect(2),10*movVect(1),10*movVect(2),'k','LineWidth',1); %Se dibuja el vector de
movimiento
h(2) = quiver(pos(1)-movVect(1),pos(2)-movVect(2),4*GTGVect(1),4*GTGVect(2),'g');
h(3) = quiver(pos(1)-movVect(1),pos(2)-movVect(2),4*AOVect(1),4*AOVect(2),'b');
h(4) = text(xPosTxt,yPosTxt,sText); %Se indica el modo en que se encuentra en la parte
superior izquierda
drawnow
pause(0.05)
for i = 1:3
    delete(h(i)); %Se borran las lineas de los puntos de paredes para que no saturen la
imagen
end
hold off

if norm(goal-pos)<=rad %Se verifica si el punto actual ha llegado al area de la meta.
    break;
end

kk = kk+1;
if kk > 1000 %Determina la cantidad de iteraciones que tendra el programa
    break;
end
end
```

ANEXO 5

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FOLLOW WALL
close all
clear all
clc

kk = 1; %Contador de Simulacion
pos = [-5,-5]; %Posicion Inicial (CAMBIABLE)
```

```
goal = [0,10]; %Meta (CAMBIABLE)
rad = 0.80; %Radio alrededor de meta (CAMBIABLE)
vel = 0.5; %Velocidad (magnitud de vector de direccion) (CAMBIABLE)
ax = [-20 20 -20 20]; %Limites del dibujo (X e Y respectivamente) (CAMBIABLE)
xPosTxt = min(ax(1:2)) + 1; %Posicion x de texto
yPosTxt = max(ax(3:4)) - 1; %Posicion y de texto

%Vectores de movimiento
GTGVect = [0 0]; %Vector que apunta hacia meta
AOVect = [0 0]; %Vector que evade obstaculos
FWVect = [0 0]; %Vector que sigue a la pared mas cercana (o al menos deberia)
movVect = [0 0]; %Vector de direccion de movimiento

%Parametros para determinar la informacion brindada por el sensor LIDAR
sect = 40; %Numero de sectores a evaluar (CAMBIABLE)
sectDeg = 360/sect;
sensInf = zeros(sect,5); %Informacion de sensor LIDAR
colFlag = 0; %Utilizado para determinar colsion

%Rectangulos de obstaculos
rect1 = rectm(-12,10,0,2); %Rectangulos de pared (CAMBIABLE)
rect2 = rectm(-14,-12,-10,2);
rect3 = rectm(10,12,-2,2);
rectMat = [rect1,rect2,rect3];

%Parametros para seguimiento de paredes
numPuntosLIDAR = 2; %Numero maximo de puntos de LIDAR empleados en el calculo.
(CAMBIABLE)
alphaW = 0.8; %Seguir pared (CAMBIABLE)
betaW = 0.3; %Mantenerse a cierta distancia de pared (CAMBIABLE)
dKept = 3; %Distancia a mantener de pared (CAMBIABLE)
dir = 0; %Direccion: Antihorario(0), Horario(1) (NO SE USA ACA)
indVect = zeros(1,numPuntosLIDAR);%vector con indices de distancias menores
distVect = zeros(1,numPuntosLIDAR);%vector con distancias menores
indLim = 0;%Numero de puntos de LIDAR que se consideran para calculos (MAXIMO
numPuntosLIDAR puntos)
matA = zeros(numPuntosLIDAR,2); %Matriz para calculo de vector
matB = zeros(numPuntosLIDAR,1);
matM = zeros(2,1);%Matriz para almacenamiento de parametros de vector
pointMat = zeros(numPuntosLIDAR,2);%Matriz para puntos estimados por sensor (5 puntos)
promPointMat = zeros(2,1); %Punto promedio entre los numPuntosLIDAR obtenidos
U = [];%Matrices para SVD para determinar si linea es vertical
S = [];
V = [];
yTrue = 0;%Determina si se plantea la ecuacion x = my + c. Si no, se utiliza y = mx + c
vecTWall = zeros(2,1);  %Vector tangente a pared
vecTWallAnt = zeros(2,1); %Vector tangente pasado, para obtener direcciones congruentes
de seguimiento.
vecPWall = zeros(2,1);  %Vector perpendicular a pared

%Parametro para evasion de obstaculos
maxDistAO = 5; %Distancia maxima tomada para obtener vector de obstacle avoidance
(CAMBIABLE)
pointsAO = zeros(sect,2); %Vectores de cada punto de LIDAR identificado
AOFact = ones(1,sect); %Factores que dan importancia a ciertos valores de LIDAR
dependiendo de su orientacion (CAMBIABLE)
factGTG = 0.8; %Factor aplicado a vector que se dirige a la meta (CAMBIABLE)
factAO = 0.2; %Factor aplicado a vector que evade obstaculos (CAMBIABLE)

%Valores utilizados para determinar cambio de modo
```

```
minDist = 0;%Distancia minima leida
detTrue = 0;%Determina si existe tal lectura
minCos = -0.9;%Se tutiliza para determinar cuanto se opone el vector GTG al AO
(CAMBIABLE)
minDistNAO = 2; %Minima distancia antes de estar en peligro de colision (AO puro)
(CAMBIABLE)
maxDistNAO = 3; %Maxima distancia a partir de la cual se puede empezar a implementar
GTG + AO (CAMBIABLE)
estado = 0; %0 = Go To Goal, 1 = Go To Goal + Avoid Obstacles, 3 = Avoid Obstacles, 4 =
Follow Wall
antEstado = 0; %Estado anterior

h = zeros(1,5); %Para dibujar vectores
sText = '';

%Dibujando graficos iniciales
figure(1)
hold on
for i=1:length(rectMat) %Dibujar rectangulos
  rectMat(i).dRect();
end
scatter(goal(1),goal(2),[],'x','r')%Dibujando meta
circle(goal(1),goal(2),rad);%Dibujando circulo de meta
scatter(pos(1),pos(2),[],'x','k') %Dibujando posicion inicial
axis(ax) %Limites de dibujo
hold off
drawnow
pause(1)

while(1)

   minDist = 0;    %Se resetean valores empleados en cada iteracion
   detTrue = 0;
   GTGVect = [0,0];
   AOVect = [0,0];
   FEVect = [0,0];

    for i = 1:sect
      [sensInf(i,1),sensInf(i,2),sensInf(i,3),sensInf(i,4),sensInf(i,5)] =
wallDistMult(rectMat,pos,sectDeg*(i-1)); %Recogiendo informacion de sensor en cada
angulo evaluado
      if sensInf(i,4)==0 %Se intenta encontrar la menor distancia
         if detTrue == 0
            minDist = sensInf(i,3);
            detTrue = 1;
         else
            if minDist>sensInf(i,3)
               minDist = sensInf(i,3);
            end
         end
      end

      if sensInf(i,4)==2 %Si posicion se encuentra dentro de rectangulo, terminar programa
por colision
         colFlag = 1;
         disp('Colision')
         break;
      end
   end
```

```
    if colFlag == 1
        break;
    end

    switch(estado) %Maquina de estados

        case 0 %GTG

            GTGVect = vel*((goal-pos)/norm(goal-pos)); %Vector desde punto actual hacia meta.
            movVect = GTGVect; %Solo GTG

            sText = 'GTG';
            antEstado = estado;

            if detTrue == 1
              if minDist <= maxDistNAO && minDist > minDistNAO %Si se acerca a un
obstaculo, se combina GTG y AO.
                 estado = 1;
              elseif minDist <= minDistNAO %Si se acerca demasiado a obstaculo, AO puro.
                 estado = 2;
              end
            end

        case 1 %GTG + AO

            GTGVect = vel*((goal-pos)/norm(goal-pos)); %Vector desde punto actual hacia meta.
            for i = 1:sect %Se evalua cada sector para determinar vector de AO
              if sensInf(i,4) == 1
                 pointsAO(i,:) = AOFact(i).*maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
%Si no se pudo leer punto en esa direccion, se asigna un vaolor arbitrario a ese vector.
              elseif sensInf(i,4) == 0
                 if sensInf(i,3)<maxDistAO %Si se leyo un punto en esa direccion se asigna un
magnitud menor a la arbitraria.
                   pointsAO(i,:) = sensInf(i,3).*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
                 else
                    pointsAO(i,:) = maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
                 end
              end

            end

            AOVect = sum(pointsAO)/sect; %Se determina el vector AO.
            AOVect = vel*(AOVect/norm(AOVect));

            movVect = factGTG.*GTGVect + factAO.*AOVect; %Se obtiene el vector de
movimiento como una suma entre lo otros vectores multiplicados por sus respectivos
factores.

            sText = 'GTG + AO';
            antEstado = estado;

            if detTrue == 1
              if minDist < minDistNAO %Si se acerca demasiado a obstaculo, AO puro.
                 estado = 2;
              elseif minDist<(dKept+0.5)&&
dot(GTGVect,AOVect)/(norm(GTGVect)*norm(AOVect))<-0.9 %Si se acerca a una pared y
los vectores GTG y AO apuntan en direcciones opuestas, se inicia el modo de seguimiento
de paredes.
                 estado = 3;
              elseif minDist > maxDistNAO %Si se aleja lo sufiviente de obstaculos, GTG puro.
```

```
            estado = 0;
        end
    end

    case 2 %AO

        GTGVect = vel*((goal-pos)/norm(goal-pos)); %Se determina vector GTG.

        for i = 1:sect %Se evalua cada sector para determinar vector de AO
            if sensInf(i,4) == 1
                pointsAO(i,:) = AOFact(i).*maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
%Si no se pudo leer punto en esa direccion, se asigna un vaolor arbitrario a ese vector.
            elseif sensInf(i,4) == 0
                if sensInf(i,3)<maxDistAO %Si se leyo un punto en esa direccion se asigna un
magnitud menor a la arbitraria.
                    pointsAO(i,:) = sensInf(i,3).*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
                else
                    pointsAO(i,:) = maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
                end
            end

        end

        AOVect = sum(pointsAO)/sect; %Se determina el vector AO.
        AOVect = vel*(AOVect/norm(AOVect));

        movVect = AOVect; %Solo AO se utiliza.

        sText = 'AO';
        antEstado = estado;

        if detTrue == 1
            if minDist>minDistAO %Una vez que se aleja lo suficiente, se decide si se siguie
una pared o se continua con el comportamiento hibrido GTG + AO

                if minDist<(dKept+0.5)&&
dot(GTGVect,AOVect)/(norm(GTGVect)*norm(AOVect))<-0.9 %Si se encuentra cerca a una
pared y vectores GTG y AO apuntan en direcciones opuestas, se sigue la pared a una
distancia dictada por dKept.
                    estado = 3;
                else %Si no, GTG + AO
                    estado = 1;
                end

            end
        end

    case 3 %FW

        GTGVect = vel*((goal-pos)/norm(goal-pos)); %Vector GTG
        %Reinicializar todas las variables tras cada iteracion
        indLim = 0; %Numero de puntos de LIDAR que se consideran para calculos
(MAXIMO 5 puntos)
        indVect = zeros(1,numPuntosLIDAR); %Vector con indices correspondientes a los
puntos elegidos
        distVect = zeros(1,numPuntosLIDAR); %Vector con distancias a puntos elegidos
        yTrue = 0; %Determina si linea predecida es vertical

        %Se ordenan las cinco (o menos) menores distancias medidas por el
        %sensor a una de las paredes
```

```
        for i = 1:sect %Se evalua cada sector para vector AO

            if sensInf(i,4) == 1
                pointsAO(i,:) = AOFact(i).*maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
            elseif sensInf(i,4) == 0
                if sensInf(i,3)<maxDistAO
                    pointsAO(i,:) = sensInf(i,3).*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
                else
                    pointsAO(i,:) = maxDistAO.*[cosd(sensInf(i,5)),sind(sensInf(i,5))];
                end
            end

            if indLim<numPuntosLIDAR %Hasta que se llenen los 5 puntos, se almacena
todos los puntos que interactuen con una pared
                if sensInf(i,4)==0 %Este valor indica el punto pertenece a una pared
                    indVect(indLim+1) = i;
                    distVect(indLim+1) = sensInf(i,3);
                    indLim = indLim + 1;
                    if indLim == numPuntosLIDAR
                        [distVect,indVect] = sortArr(distVect,indVect); %Esta funcion ordena el
vector de distancia de menor a mayor y realiza los mismos cambios en el vector de indices.
                    end
                end
            else
                if sensInf(i,4)==0 && distVect(end)>sensInf(i,3) %Reemplaza el punto con la
mayor distancia si el nuevo punto tiene una menor
                    distVect(end) = sensInf(i,3);
                    indVect(end) = i;
                    [distVect,indVect] = sortArr(distVect,indVect); %Reordena ambos vectores de
nuevo para poner la mayor distancia al final
                end
            end
        end
        if indLim<numPuntosLIDAR %Si se llenaron menos de 5 espacios, se ordena de
menor a mayor puesto que no se hizo antes
            [distVect,indVect] = sortArr(distVect,indVect);
        end
        distVect = fliplr(distVect);%Se ordenan ambos vectores de mayor a menor por si
distancias 0 se encuentran presentes.
        indVect = fliplr(indVect);


        for i = 1:indLim %Se estima los puntos en las paredes utilizando los angulos
evaluados y las distancias encontradas
            pointMat(i,:) = [distVect(i)*cosd(sectDeg*(indVect(i)-1))
distVect(i)*sind(sectDeg*(indVect(i)-1))];
            matA(i,:) = [pointMat(i,1) 1]; %Matrices para armar polinomios de grado 1 (y = mx +
c)
            matB(i,1) = pointMat(i,2);
        end

        [U,S,V] = svd(matA(1:indLim,:)); %SVD para determinar si la matriz A es singular (si
linea es casi vertical)
        if log(max(sum(S))/min(sum(S)))>10 %Esto determina lo establecido anteriormente
            yTrue = 1; %Se plantea el siguiente polinomio: x = my + c
            for i = 1:indLim
                matA(i,:) = [pointMat(i,2) 1];
                matB(i,1) = pointMat(i,1);
            end
            [U,S,V] = svd(matA(1:indLim,:));
```

```
            if log(max(sum(S))/min(sum(S)))>10 %Se evalua de nuevo en caso de que haya
algun problema
                disp('Error con SVD') %Si esto ocurre, lo mejor es salir del programa y evaluar
esta ultima iteracion para ver que sucedio
                break;
            end
        end

        matM = matA(1:indLim,:)\matB(1:indLim); %Se obtiene la linea mas cercana a los
puntos elegidos por medio de metodo least-square fitting
        promPointMat = sum(pointMat,1)/indLim; %Punto promedio de los puntos elegidos.
Se utilizara para determinar la direccion de giro

        %Se obtiene un vector que apunta en la direccion de la linea obtenida,
        %es decir, un vector paralelo (tagente) a la pared (aproximada)
        if yTrue == 0 % y = mx + c
            vecTWall = [1 matM(1)];
        else % x = my + c
            vecTWall = [matM(1) 1];
        end

        if antEstado ~= 3 %Tras cambio de estado, se decide elegir una direccion
congruente con el primer vector GTG obtenido.
            if dot(vecTWall,GTGVect)<0
                vecTWall = -vecTWall; %Si no, el vector cambia de direccion.
            end
        else %En iteraciones siguientes, se sigue la direccion congruente con las
direcciones anteriores.
            if dot(vecTWall,vecTWallAnt)<0
                vecTWall = -vecTWall; %Si no, el vector cambia de direccion.
            end
        end

        vecTWall = vecTWall/norm(vecTWall); %Se normaliza el vector tangente.
        vecTWallAnt = vecTWall;

        vecPWall = pointMat(indLim,:) - dot(pointMat(indLim,:),vecTWall)*vecTWall; %Se
obtiene un vector perpendicular a la pared a partir del punto mas cercano y el vector
tangente.
        vecPWall = vecPWall - dKept*(vecPWall/norm(vecPWall)); %Se intenta que el vector
prependicular permita mantener la distancia establecida en dKept a las paredes.

        FWVect = alphaW*vecTWall + betaW*vecPWall; %El vector de movimiento es una
combinacion entre  los vectores tangente y perpendicular. La influencia de cada uno se
determina por medio de alpha y beta.
        FWVect = vel*(FWVect/norm(FWVect)); %El valor vel determina la magnitud del
vector de movimiento.

        AOVect = sum(pointsAO)/sect; %Vector AO
        AOVect = vel*(AOVect/norm(AOVect));

        movVect = FWVect; %Solo se sigue pared. Vectores AO y GTG se utilizan para
determinar si se debe cambiar de modo.

        sText = 'FW';
        antEstado = estado;

        if detTrue == 1 %Si los Vectores GTG y AO se encuentran apuntando en la misma
direccion, asi como el vector FW y GTG, se cambia a modo GTG + AO
```

```
            if dot(GTGVect,AOVect)/(norm(GTGVect)*norm(AOVect))>=0.1 &&
dot(GTGVect,FWVect)/(norm(GTGVect)*norm(FWVect))>=0.1
                estado = 1;
            end
        end

    end

    pos = pos + movVect; %Se actualiza la posicion basada en el vector de movimiento.

    hold on
    scatter(pos(1),pos(2),[],'x','b') %Se dibuja el nuevo punto.
    h(1) = quiver(pos(1)-movVect(1),pos(2)-
movVect(2),10*movVect(1),10*movVect(2),'k','LineWidth',1); %Se dibuja el vector de
movimiento
    h(2) = quiver(pos(1)-movVect(1),pos(2)-movVect(2),4*GTGVect(1),4*GTGVect(2),'g');
%Se dibuja el vector GTG
    h(3) = quiver(pos(1)-movVect(1),pos(2)-movVect(2),4*AOVect(1),4*AOVect(2),'b'); %Se
dibuja el vector AO
    h(4) = quiver(pos(1)-movVect(1),pos(2)-movVect(2),4*FWVect(1),4*FWVect(2),'m'); %Se
dibuja el vector FW
    h(5) = text(xPosTxt,yPosTxt,sText); %Se indica el modo en que se encuentra en la parte
superior izquierda
    drawnow
    pause(0.05)
    for i = 1:5
        delete(h(i)); %Se borran las lineas de los puntos de paredes para que no saturen la
imagen
    end
    hold off

    if norm(goal-pos)<=rad %Se verifica si el punto actual ha llegado al area de la meta.
        break;
    end

    kk = kk+1;
    if kk > 1000 %Determina la cantidad de iteraciones que tendra el programa
        break;
    end

end
```

ANEXO 6

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  #Search and avoid program changed to be used for testing in an Odroid coupled to a
    LIDAR device and a Pixhawk Flight controller in a controlled environment.
4.
5.  from sweeppy import Sweep #For Sweep LIDAR sensor
6.  from dronekit import connect, VehicleMode, LocationGlobal, LocationGlobalRelative #Li
    brary for communication with Pixhawk (sends mavlink messages)
7.  from pymavlink import mavutil # Needed by dronekit for mavlink communication
8.  import numpy as np #Numpy for mathematical calculations and vector calculus.
    Identical to matlab libraries.
9.  import numpy.matlib as M #Numpy library for handling matrices and matrix calculations.
10.
```

```
from multiprocessing import Process, Queue, Lock #Program will handle multiple processes (Odroid has multiple processors) which will communicate with each other through queues.
import sys, os, time, signal

R_EARTH = 6371e3        #Earth radius
main_Interrupt = False #Will be used to exit the program by pressing ctrl + c
#POS_INI = np.array([-12.065401,-77.080415]) # Just used as reference, don't need to change this
#POS_FIN = np.array([-12.065013,-77.078827])

################################################JJ SAVE LOGS#################################################################
#f= open("log.txt","w")
#f.write("LOG DE VUELO"+'\n')
#f.write("Modo   Distancia a objetivo   Actual direccion   Minima distancia"+'\n')
#f.close()

#############################################################################################################END JJ#####################
def sortArr(V,D):
        #Will be used to order the closest LIDAR points along their associated distance and degree.
        #V = Distance vector to be sorted
        #D = Degree vector to be sorted
        #Varr = Distance vector sorted from smallest to highest distance
        #Darr = Degree vector sorted to maintain a one on one correspondance with the Varr vector

        indVals = np.argsort(V)
        Varr = np.sort(V)
        Darr = D[indVals]

        return Varr,Darr

def normC(arr):
        #Calculates the magnitude of an array
        return np.sqrt(np.sum(arr*arr))

def headingT(heading):
        #Change vehicle heading to GPS bearing calculated in distBrngGlobal
        if heading > 180.0:
                hd = np.deg2rad(heading - 360.0)
        else:
                hd = np.deg2rad(heading)
        return hd
        #0<=heading<=360
        #-180<=hd<=180

def distBrngGlobal(loc1,loc2):
        #Calculates distance and bearing between to coordinates
        #Based on distance and bearing calculations in http://www.movable-type.co.uk/scripts/latlong.html (in meters and radians respectively)
        phi_1 = np.deg2rad(loc1.lat)
        phi_2 = np.deg2rad(loc2.lat)
        phi_delta = np.deg2rad(loc2.lat - loc1.lat)
        lambda_delta = np.deg2rad(loc2.lon - loc1.lon)

```

```
63.         am = (np.sin(phi_delta/2)*np.sin(phi_delta/2) +
64.               np.cos(phi_1) * np.cos(phi_2) *
65.               np.sin(lambda_delta/2)*np.sin(lambda_delta/2))
66.
67.         cm = 2*np.arctan2(np.sqrt(am),np.sqrt(1-am))
68.         dist = R_EARTH*cm
69.
70.         yVal = np.sin(lambda_delta)*np.cos(phi_2)
71.         xVal = (np.cos(phi_1)*np.sin(phi_2) -
72.                     np.sin(phi_1)*np.cos(phi_2)*np.cos(lambda_delta))
73.         brng = np.arctan2(yVal,xVal)
74.
75.         return dist,brng
76.
77. def decisionMaker(quadDev,q_Dist,q_Vect,l_Dist,l_Vect):
78.
79.         #Process in charge of deciding the direction and magnitude of the quadcopter
    velocity vector based on LIDAR measurements and current GPS position.
80.         #Communicates with wallDetector process through q_Dist and q_Vect queues.
    l_Dist and l_Vect are locks used to extract information from the queues.
81.         '''**************************************************************************************************
    ************************************************************'''
82.         R_EARTH = 6371e3              #Earth radius in meters
83.         MIN_DIST_NAO = 2.0      #Minimum distance before engaging pure AO behavior
84.         MAX_DIST_NAO = 5.0            #Maximum distance to engage GTG + AO behavior
85.         D_KEPT = 3.0                  #Distance kept from walls in FW
    behavior       DISTANCIA A MANTENER DE UNA PARED EN METROS
86.         MAX_PERP = -0.75                    #Used to determine angle between GTG and
    AO behaviors to change to FW state. Equal to COS(MAX_ANGLE).
87.         MIN_PERP = 0.15
88.
89.         GTGF = 0.8                    #Presence factor of GTG vector in GTG + AO state
90.         AOF = 0.3                     #Presence factor of AO vector in GTG + AO state
91.
92.         droneSpeed = 0.5              #Drone speed. Has to be lower than 0.5 for testing
    purposes. ESTA ES LA VELOCIDAD DEL DRON EN M/S
93.         currYaw = 0.0                 #Current orientation fo drone
94.         brng2GoalGPS = 0.0            #Calculated through GPS locations
95.         brng2Goal = 0.0               #Calculated taking into account local heading
96.         dist2Goal = 500.0             #Distance from goal
97.         minDistObs = 500.0            #Minimum distance to obstacle
98.
99.         alphaW = 0.8                  #AlphaW and BetaW are values that control the wall
    following behavior
100.            betaW = 0.3
101.            dKept = 3.0                     #Distance kept from walls DISTANCIA A
    MANTENER DE PARED EN METROS
102.
103.            gConn = True                  #Variable used to exit program if vehicle connection
    fails
104.            #vehicle = connect(device, baud = 115200, wait_ready=True)
105.
106.
107.
108.            '''**************************************************************************************************
    ************************************************************'''
109.
110.            L2Veh = np.array([1,-1]) #Converts LIDAR vector into vehicle compatible
    vector
111.
```

```
112.            GTGVect = np.zeros(2)               #Vector points to goal
113.            AOVect = np.zeros(2)               #Vector points away from closest object
114.            vecTWall = np.zeros(2)             #Vector tangent to wall
115.            vecTWallAnt = np.zeros(2)            #Keeps previous vecTWAll value
116.            vecPWall = np.zeros(2)             #Vector perpendicular to wall
117.            closePVec = np.zeros(2)             #Vector to closest point for vecPWall
    calculation
118.            FWVect = np.zeros(2)               #Vector combines vecTWall and vecPWall
    through alphaW and betaW values
119.            MovVect = np.zeros(2)              #Vector determines the quadcopter velocity
    vector for 1 second
120.
121.            currState = 0   #Current program state. Dictates the behavior to follow. 0 =
    GTG, 1 = GTG + AO, 2 = AO, 3 = FW
122.            lastState = 0
123.            modeS = 'GTG'   #Used to print in console current state
124.
125.
126.            try:
127.                print 'Connecting to vehicle to: {0}'.format(quadDev)
128.                #Try connecting to quad.
129.                try:
130.                    vehicle = connect(quadDev, wait_ready=True, baud=57600)
131.                    vehicle.wait_ready('autopilot_version')
132.                except:
133.                    gConn = False
134.
135.                #If connection to quad fails, close console.
136.                while not gConn:
137.                    print 'Wrong QUAD device, waiting for ctrl + c.'
138.                    time.sleep(1)
139.
140.                print 'Connected to drone!'
141.                time.sleep(0.5)
142.
143.                #Functions are declared in the scope of the process to use variable
    "vehicle" without having to define it in each one.
144.
145.                def arm_and_takeoff(aTargetAltitude):
146.                    """
147.                    Arms vehicle and fly to aTargetAltitude.
148.                    """
149.                    print 'Basic pre-arm checks'
150.                    #Don't let the user try to arm until autopilot is ready
151.                    while not vehicle.is_armable:
152.                        print 'Waiting for vehicle to initialise...'
153.                        time.sleep(1)
154.
155.                    print 'Arming motors'
156.                    # Copter should arm in GUIDED mode
157.                    vehicle.mode = VehicleMode('GUIDED')
158.                    while not vehicle.mode.name=='GUIDED':  #Wait until mode has
    changed
159.                        print " Waiting for mode change ..."
160.                        time.sleep(1)
161.                    vehicle.armed = True
162.
163.                    while not vehicle.armed:
164.                        print 'Waiting for arming...'
```

```
165.                         time.sleep(1)
166.
167.                     print 'Taking off!'
168.                     vehicle.simple_takeoff(aTargetAltitude) # Take off to target altitude.
169.                     #TODO: Can be changed to allow changing the takeoff acceleration
    and velocity.
170.
171.                     # Wait until the vehicle reaches a safe height before processing the
    goto (otherwise the command
172.                     #  after Vehicle.simple_takeoff will execute immediately).
173.                     while True:
174.                         print 'Altitude: ', vehicle.location.global_relative_frame.alt
175.                         if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.
    95: #Trigger just below target alt.
176.                             print 'Reached target altitude'
177.                             break
178.                         time.sleep(1)
179.
180.                 def send_ned_velocity(velocity_x, velocity_y, velocity_z):
181.                     #Send local velocity to vehicle via mavlink message
182.                     msg = vehicle.message_factory.set_position_target_local_ned_enco
    de(
183.                         0,       # time_boot_ms (not used)
184.                         0, 0,    # target system, target component
185.                         mavutil.mavlink.MAV_FRAME_BODY_OFFSET_NED, # frame
186.                         0b0000111111000111, # type_mask (only speeds enabled)
187.                         0, 0, 0, # x, y, z positions (not used)
188.                         velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
189.                         0, 0, 0, # x, y, z acceleration (not supported yet, ignored in
    GCS_Mavlink)
190.                         0, 0)    # yaw, yaw_rate (not supported yet, ignored in
    GCS_Mavlink)
191.                     vehicle.send_mavlink(msg)
192.
193.                 print 'Beginning...'
194.                 '''*************************************************************************************
    ********************************************************************'''
195.                 tgtAlt = 2      #Operation altitude. ESTA ES LA ALTITUD DESEADA DE
    OPERACION. SE PUEDE CAMBIAR DEPENDIENDO DEL CRITERIO DEL
    OPERADOR
196.                 '''*************************************************************************************
    ********************************************************************'''
197.                 #Arm and takeoff to 2.5 meters
198.
199.
200.                 arm_and_takeoff(tgtAlt)
201.
202.                 print 'Starting LIDAR...'
203.                 #wallDetector will start reading points after it received a variable through
    q_Dist queue
204.                 l_Dist.acquire()
205.                 q_Dist.put(0)
206.                 l_Dist.release()
207.
208.                 #It takes a while for the LIDAR to start rotating, so maybe it would be
    preferable to increase sleeping time or else the quad will begin to move before LIDAR
    readings are available.
209.                 time.sleep(1)
210.
211.                 print 'Movement start'
```

```
212.                vehicle.groundspeed = droneSpeed
213.                vehicle.airspeed = droneSpeed
214.
215.                currLoc = LocationGlobalRelative(0,0,tgtAlt)                    #G
    ets current GPS location
216.                #goalLoc = LocationGlobalRelative(-12.065013,-
    77.078827,tgtAlt)       #GPS Goal Location
217.                #goalLoc = LocationGlobalRelative(-12.064028, -77.080434,tgtAlt)
218.                goalLoc = LocationGlobalRelative(-12.072055, -
    77.081965,tgtAlt)       #Sets goal coordinates. Change this value depending on current
    mission.
219.
220.                time.sleep(1)
221.                t2 = time.time()       #Used for performance purposes.
222.
223.                while True:
224.                    currYaw = headingT(vehicle.heading)
            #Transforms droneKit heading range (0,360) to regular GPS heading range (-
    180,180)
225.                    currLoc = vehicle.location.global_relative_frame                #
    Gets current drone coordinates
226.                    dist2Goal,brng2GoalGPS = distBrngGlobal(currLoc,goalLoc)
     #Calculates distance and bearing to goal
227.
228.                    #Goal is considered to have been reached if quad is 5 meters from it.
229.                    if dist2Goal < 5.0:
230.                        send_ned_velocity(0,0,0)
231.                        print 'Goal reached!'
232.                        time.sleep(1)
233.                        break
234.
235.                    brng2Goal = brng2GoalGPS -
    currYaw                                     #Difference between vehicle heading and
    heading towards goal
236.                    GTGVect = np.array([np.cos(brng2Goal),np.sin(brng2Goal)])
     #Unit vector pointing towards goal
237.
238.                    #Get minimum distance to closest obstacle.
239.                    l_Dist.acquire()
240.                    minDistObs = q_Dist.get()
241.                    q_Dist.put(minDistObs)
242.                    l_Dist.release()
243.
244.                    if currState != 0:
245.                        #get current vecTWall, closePVec and AOVect as calculated by
    wallDetector process
246.                        l_Vect.acquire()
247.                        vecTWall, closePVec, AOVect = q_Vect.get()
248.                        q_Vect.put([vecTWall,closePVec,AOVect])
249.                        l_Vect.release()
250.                        AOVect = L2Veh*AOVect   #Simple transformation into regular
    coordinates.
251.
252.                    #State Machine depending on current behavior or state
253.                    if currState == 0:          #GTG
254.
255.                        MovVect = droneSpeed*GTGVect
256.                        #lastState = currState
257.                        modeS = 'GTG'
258.
```

```
259.                            #Conditions for state change
260.                            if (minDistObs <= MAX_DIST_NAO) and (minDistObs > MIN_DI
    ST_NAO):
261.                                currState = 1
262.                            elif (minDistObs <= MIN_DIST_NAO):
263.                                currState = 2
264.
265.                        elif currState == 1:    #GTG + AO
266.
267.                            MovVect = droneSpeed*(GTGF*GTGVect + AOF*AOVect)
268.                            #lastState = currState
269.                            modeS = 'GTG + AO'
270.
271.                            #Conditions for state change. lastState is only used when
    changing to Follow Wall behavior.
272.                            if minDistObs < MIN_DIST_NAO:
273.                                currState = 2
274.                            elif (minDistObs < (D_KEPT
    + 0.25)) and ((np.inner(GTGVect,AOVect)/(normC(GTGVect)*normC(AOVect))) < MAX_
    PERP):
275.                                currState = 3
276.                                lastState = 1
277.                            elif minDistObs > MAX_DIST_NAO:
278.                                currState = 0
279.
280.                        elif currState == 2:    #AO
281.
282.                            MovVect = droneSpeed*AOVect
283.                            #lastState = currState
284.                            modeS = 'AO'
285.
286.                            #Conditions for state change. lastState is only used when
    changing to Follow Wall behavior.
287.                            if minDistObs > MIN_DIST_NAO:
288.                                if (minDistObs < (D_KEPT
    + 0.25)) and ((np.inner(GTGVect,AOVect)/(normC(GTGVect)*normC(AOVect))) < MAX_
    PERP):
289.                                    currState = 3
290.                                    lastState = 2
291.                                else:
292.                                    currState = 1
293.
294.                        elif currState == 3:    #FW
295.
296.                            #Follow wall will maintain a clockwise or counter-clockwise
    direction to surround obstacle depending on the GTGVect of the first time it enters this
    state
297.                            if lastState != 3:
298.                                if np.dot(vecTWall,GTGVect)<0 :
299.                                    vecTWall = -vecTWall
300.                                lastState = 3
301.                            else:
302.                                if np.dot(vecTWall,vecTWallAnt)<0 :
303.                                    vecTWall = -vecTWall
304.
305.                            #if ((np.cross(vecTWall,promPointMat)>=0 and
    dir==1)or(np.cross(vecTWall,promPointMat)<0 and dir==0)):
306.                            #       vecTWall = -vecTWall
307.
308.                            vecTWallAnt = vecTWall
```

```
309.                            vecTWall = vecTWall/normC(vecTWall)
310.
311.
312.                            #vecPWall = pointMat[indLim-1,:] - np.dot(pointMat[indLim-
    1,:],vecTWall)*vecTWall
313.                            vecPWall = closePVec - np.dot(closePVec,vecTWall)*vecTWall
314.                            nvecPWall = -L2Veh*vecPWall     #Simple transformation into
    regular coordinates for state change conditions.
315.                            vecPWall = vecPWall - dKept*(vecPWall/normC(vecPWall))
316.
317.                            FWVect = alphaW*vecTWall + betaW*vecPWall
318.                            FWVect = L2Veh*(FWVect/normC(FWVect)) #Simple
    transformation into regular coordinates.
319.
320.                            MovVect = droneSpeed*FWVect
321.                            #lastState = currState
322.                            modeS = 'FW'
323.
324.                            #nvecPWall = L2Veh*vecPWall
325.
326.                            #if
    (np.dot(GTGVect,AOVect)/(normC(GTGVect)*normC(AOVect)) >= 0.1) and
    (np.dot(GTGVect,FWVect)/(normC(GTGVect)*normC(FWVect)) >= 0.1):
327.                            #if
    (np.dot(GTGVect,AOVect)/(normC(GTGVect)*normC(AOVect)) >= MIN_PERP) and
    (minDistObs > D_KEPT-0.5):
328.                            #Basically, if a certain distance is kept from the perceived wall
    and GTGVect is somewhat aligned with a vector that points away from the closest
    obstacle.
329.                            if (np.dot(GTGVect,nvecPWall)/(normC(GTGVect)*normC(nvec
    PWall)) >= MIN_PERP) and (minDistObs > D_KEPT-0.5):
330.                                    currState = 1
331.                            elif minDistObs > MAX_DIST_NAO: #If, somehow, the obstacle
    disappears, the quad should head straight to goal.
332.                                    currState = 0
333.                        '''*****************************************************************************
    *************************************************************************'''
334.                        #SIEMPRE SE VA A IMPRIMIR EL MODO DE VUELO, LA
    DISTANCIA A LA META, LA DIRECCION ACTUAL SEGUIDA Y LA DISTANCIA AL
    PUNTO MAS CERCANO
335.
336.                        print 'Mode: {0}, Dist2Goal: {1} m, CurrDir: {2} deg, MinDist: {3}
    m'.format(modeS,dist2Goal,np.arctan2(MovVect[0],MovVect[1])* 180 / np.pi,minDistObs)
337.
338.    ##################################################JJ****SAVE
    LOGS#############################################################################
    ######################################
339.                        # f= open("log.txt","a")
340.                        # f.write('\n' + "Linea %d\t" % (jindex))
341.                        ## f.write('Dist rem: {} m,  BrngDiff: {}
    deg'.format(dist2Goal,brng2Goal))
342.                        ##  f.write('Mode: {}, Dist2Goal: {} m, MinDist: {}
    m'.format(modeS,dist2Goal,minDistObs))
343.                        # f.write('Mode: {0}, Dist2Goal: {1} m, CurrDir: {2} deg, MinDist: {3}
    m'.format(modeS,dist2Goal,np.arctan2(MovVect[0],MovVect[1])* 180 /
    np.pi,minDistObs))
344.                        # jindex = jindex+1
345.                        # f.close()
```

```
346.    ##################################################################################################################END JJ########
347.
348.
349.                    #A speed command is sent every second
350.                    while time.time()-t2 < 1.0:
351.                        time.sleep(0.01)
352.                    send_ned_velocity(MovVect[0],MovVect[1],0)
353.                    t2 = time.time()
354.
355.                time.sleep(5)
356.                #After reaching the goal, vehicle will land.
357.                vehicle.mode = VehicleMode('LAND')
358.                print 'Landing...'
359.                while vehicle.armed:
360.                    time.sleep(1)
361.                print 'Landing successful. Vehicle disarmed. Closing program.'
362.                vehicle.close()
363.
364.                print "QUAD closed, waiting for ctrl + c to finish program."
365.                while True:
366.                    time.sleep(1)
367.
368.            except KeyboardInterrupt:
369.                print('Interrupt received in decisionMaker!')
370.
371.                try:
372.                    vehicle.close()
373.                except:
374.                    print('No vehicle, still exiting.')
375.
376.            finally:
377.    #####JJ****SAVE LOGS
378.                #text_file.close()
379.    ####END JJ///////
380.                print('Ending decisionMaker!')
381.
382.    def wallDetector(lidDev,motorHz,sampleHz,numLIDAR,q_Dist,q_Vect,l_Dist,l_Vect):
383.
384.            #Process in charge of processing LIDAR information and estimating objects as obstacles and walls. Only a single wall can be estimated for now.
385.            #Communicates with decisionMaker process through q_Dist and q_Vect queues. l_Dist and l_Vect are locks used to put information into the queues.
386.            '''************************************************************************************************************************'''
387.            MIN_DIST = 2.0          #Minimal distance kept from walls. QUE SEAN CONGRUENTE CON LOS VALORES DE ARRIBA
388.            MAX_DIST = 5.0  #Maximum distance the sensors take into account from sensors (to not overwhelm calculations). QUE SEA CONGRUENTE CON LOS VALORES DE ARRIBA
389.            MIN_SIGNAL = 128        #Minimal signal strength to take measurement into account.
390.            '''************************************************************************************************************************'''
391.
392.            #dKept = 3.0                #Distance kept from walls
393.            #alpha = 0.8                #Alpha and Beta are values that control de wall following behavior
```

```
394.            #beta = 0.3
395.            #dir = 0                    #0 = Counterclockwise, 1 = Clockwise
396.
397.            #U = None                   #Matrices for SVD to determine whether the line is
    perpendicular with respect to current heading
398.            S = None
399.            #V = None
400.
401.            matA = M.zeros((numLIDAR,2))    #Matrices for wall vector calculation
402.            matB = M.zeros((numLIDAR,1))
403.            matM = M.zeros((2,1))           #Matrix for line parameters of the wall line
404.
405.            pointMat = M.zeros((numLIDAR,2))        #Matrix with estimated wall points in
    XY relative coordinates
406.            promPointMat = M.zeros((1,2))           #Average of chosen points
407.            yTrue = False                           #Determines whether line is y = mx +
    c (False) or x = my + c (True)
408.
409.            #vecTWall = np.zeros(2);             #Vector tangent to wall
410.            #vecPWall = np.zeros(2);             #Vector perpendicular to wall
411.
412.            time.sleep(0.01)
413.
414.            try:
415.                with Sweep(lidDev) as sLid:         #Always create a LIDAR object like
    this o prevent issues with the library.
416.
417.                    sLid.stop_scanning()        #Stops LIDAR scanning.
418.                    sLid.set_motor_speed(0)     #Stops LIDAR rotation.
419.
420.                    print 'Motor speed: {0} Hz'.format(sLid.get_motor_speed())
421.                    print 'Sample rate: {0} Hz'.format(sLid.get_sample_rate())
422.
423.                    TWVect = np.zeros(2)        #Initiates all vectors to zeros.
424.                    AOVect = np.zeros(2)
425.                    TWDummy = np.zeros(2)       #Dummy vectors used to keep
    queues full.
426.                    closePDummy = np.zeros(2)
427.                    AODummy = np.zeros(2)
428.
429.                    l_Vect.acquire()
430.                    q_Vect.put([TWDummy,closePDummy,AODummy])   #Initial
    vectors in queue
431.                    l_Vect.release()
432.
433.                    print 'LIDAR OK. Waiting for instructions...'
434.                    #Wait for queue signal from decisionMaker process.
435.                    while q_Dist.empty():
436.                        time.sleep(0.1)
437.
438.                    l_Dist.acquire()
439.                    q_Dist.get()
440.                    q_Dist.put(MAX_DIST)        #Initial maximum measured distance
    sent to decisionMaker
441.                    l_Dist.release()
442.
443.                    #Setting LIDAR motor rotation speed and sample frequency.
444.                    if (motorHz != 0):
445.                        sLid.set_motor_speed(motorHz)
446.                        print 'New Motor speed: {0} Hz'.format(sLid.get_motor_speed())
```

```
447.                    else:
448.                        sLid.set_motor_speed(4) #This will start motor rotation.
449.
450.                    if (sampleHz != 0):
451.                        sLid.set_sample_rate(sampleHz)
452.                        print 'New Sample rate: {0} Hz'.format(sLid.get_sample_rate())
453.                    else:
454.                        sLid.set_sample_rate(500)
455.
456.                    print 'Start scanning!'
457.                    sLid.start_scanning()   #This will start motor scanning. This will take a
    couple of seconds.
458.
459.                    print 'Getting generator...'
460.                    s = sLid.get_scans()    #This generator will keep spawning LIDAR
    measurements until programs ends.
461.                    print 'Generator obtained!'
462.
463.                    #t1 = time.time()
464.
465.                    for scanLid in s:
466.                        #indVect = np.zeros(numLIDAR)                    #vector with
    indices corresponding to shortest distances
467.                        distVect = np.zeros(numLIDAR)                    #vector with
    shortest distances
468.                        degVect = np.zeros(numLIDAR)                     #vector with
    degrees corresponding to shortest distances
469.                        indLim = 0                                       #number
    of LIDAR points obtained for calculation (MAX = numLIDAR)
470.                        yTrue =False                                     #Determines
    whether wall is perpendicular to current heading
471.                        minDist = MAX_DIST                               #Minimu
    m distance obtained from single scan
472.                        AOPoints = np.zeros(2)                           #Used to
    determine vector for Avoid Obstacle behavior
473.
474.                        scanNum = len(scanLid.samples)                   #The number
    of samples changes during each scan and depends on rotation and scanning frequency
475.                        #Ordering "numLIDAR" or less number of points in order
    depending of distance, taking into account MAX_DIST and MIN_DIST
476.                        #Also, sum measured points to AOPoints to form a vector
    directed away from the closest obstacle.
477.                        for lSample in scanLid.samples:
478.                            sampDist = (lSample.distance/float(100))     #Sample
    distance in meters
479.                            sampAng = lSample.angle/float(1000)          #Sampl
    e angle in degrees with respect to sensor front (goes from 0 to 360 in CCW fashion)
480.
481.                            if (lSample.signal_strength > MIN_SIGNAL):   #Only take
    into account measurements with decent signal strength
482.
483.                                if sampDist < MAX_DIST:
484.                                    AOPoints = AOPoints +
    np.array([sampDist*np.cos(np.deg2rad(sampAng)),sampDist*np.sin(np.deg2rad(sampA
    ng))])
485.                                else:
486.                                    AOPoints = AOPoints +
    np.array([MAX_DIST*np.cos(np.deg2rad(sampAng)),MAX_DIST*np.sin(np.deg2rad(sa
    mpAng))])
487.
```

```
488.                                    if ((indLim < numLIDAR) and (sampDist < MAX_DIST
    )):
489.                                        #indVect[indLim] = i
490.                                        degVect[indLim] = sampAng
491.                                        distVect[indLim] = sampDist
492.                                        indLim = indLim + 1
493.                                        if indLim == numLIDAR:
494.                                            distVect,degVect = sortArr(distVect,degVect
    )
495.                                    elif ((indLim >= numLIDAR) and (sampDist < distVect[
    -1])):
496.                                        degVect[-1] = sampAng
497.                                        distVect[-1] = sampDist
498.                                        distVect,degVect = sortArr(distVect,degVect)
499.
500.                                    if sampDist<minDist:
501.                                        minDist = sampDist
502.
503.                                else:
504.                                    AOPoints = AOPoints +
    np.array([MAX_DIST*np.cos(np.deg2rad(sampAng)),MAX_DIST*np.sin(np.deg2rad(sa
    mpAng))])
505.
506.                                #Previous algorithm
507.                                '''if indLim < numLIDAR:       #Points will be added until
    maximum is reached
508.                                    if (lSample.signal_strength > MIN_SIGNAL) and
    (sampDist < MAX_DIST):
509.                                        #indVect[indLim] = i
510.                                        degVect[indLim] = lSample.angle/float(1000)
511.                                        distVect[indLim] = sampDist
512.                                        if sampDist<minDist:
513.                                            minDist = sampDist
514.                                        indLim = indLim + 1
515.                                        if indLim == numLIDAR:
516.                                            distVect,degVect =
    sortArr(distVect,degVect)
517.                                else:
518.                                    if (lSample.signal_strength > MIN_SIGNAL) and
    (sampDist) < distVect[-1]):
519.                                        degVect[-1] = lSample.angle/float(1000)
520.                                        distVect[-1] = lSample.distance/float(100)
521.                                        distVect,degVect = sortArr(distVect,degVect)
522.                                        if sampDist<minDist
523.                                            minDist = sampDist'''
524.
525.                            #Sort distance and degree vector if maximum number of
    readable samples hasn't been reached
526.                            if indLim < numLIDAR:
527.                                distVect,degVect = sortArr(distVect,degVect)
528.
529.                            #Sent minimum distance acquired to decisionMaker process
530.                            l_Dist.acquire()
531.                            q_Dist.get()
532.                            q_Dist.put(minDist)
533.                            l_Dist.release()
534.
535.                            #Get mean AO vector from AOPoints
536.                            AOVect = AOPoints/float(scanNum)
537.
```

```
538.                        #Send AOVect to decisionMaker process and keep other
    vectors in queue
539.                        l_Vect.acquire()
540.                        TWDummy, closePDummy, AODummy = q_Vect.get()
541.                        q_Vect.put([TWDummy,closePDummy,AOVect])
542.                        l_Vect.release()
543.
544.                        #If no measurements with enough signal strength were
    obtained, continue to next sample of measurements
545.                        if indLim == 0:
546.                            time.sleep(0.01)
547.                            continue
548.
549.                        distVect = distVect[::-1]           #Flip both distance and degree
    vectors (lists)
550.                        degVect = degVect[::-1]
551.
552.                        #pointMat creates points from distance and angle vectors with
    sufficient signal strength.
553.                        #Matrices A and B are created to estimate a wall from the read
    points to solve Ax = B
554.                        for ind in range(indLim):
555.                            pointMat[ind,:] = [distVect[ind]*np.cos(np.deg2rad(degVect[i
    nd])),distVect[ind]*np.sin(np.deg2rad(degVect[ind]))]
556.                            matA[ind,:] = [pointMat[ind,0],1]
557.                            matB[ind,0] = pointMat[ind,1]
558.
559.                        #SVD decomposition helps determine if matrix A is proper for
    the algorithm to use
560.                        S = np.linalg.svd(matA[range(indLim),:],full_matrices=False,com
    pute_uv=False)
561.                        if np.log10(S[0]/float(S[-1]))>10:     #If not true, polynomial to
    estimate wall is y = mx + c, with m and c as variables.
562.                            yTrue = True          #Polynomial is x = my + c
563.                            for ind in range(indLim):
564.                                matA[ind,:] = [pointMat[ind,1],1]
565.                                matB[ind,0] = pointMat[ind,0]
566.                            S = np.linalg.svd(matA[range(indLim),:],full_matrices=False
    ,compute_uv=False)
567.                            if np.log10(S[0]/float(S[-1]))>10:
568.                                print 'Problema con SVD.'
569.                                continue
570.
571.                        matM = np.linalg.lstsq(matA,matB)[0]    #Obtaining m and c
572.                        #promPointMat =
    pointMat.sum(axis=0,dtype='float')/float(indLim)
573.
574.                        #Vector tangent to wall created using m from the polynomial that
    estimates the wall linear equation
575.                        if yTrue == False:
576.                            vecTWall = np.array([1,matM[0]])
577.                        else:
578.                            vecTWall = np.array([matM[0],1])
579.
580.                        #This vector is sent to decisionMaker process.
581.                        l_Vect.acquire()
582.                        TWDummy, closePDummy, AODummy = q_Vect.get()
583.                        q_Vect.put([vecTWall,np.squeeze(np.asarray(pointMat[indLim-
    1,:])),AODummy])
584.                        l_Vect.release()
```

```
585.
586.                             #Previous algorithm. No longer used since FWVect requires
    information in decisionMaker process to be completely defined.
587.                             #if ((np.cross(vecTWall,promPointMat)>=0 and
    dir==1)or(np.cross(vecTWall,promPointMat)<0 and dir==0)):
588.                             #       vecTWall = -vecTWall
589.                             #vecTWall = vecTWall/normC(vecTWall)
590.
591.                             #vecPWall = pointMat[indLim-1,:] - np.dot(pointMat[indLim-
    1,:],vecTWall)*vecTWall
592.                             #vecPWall = vecPWall - dKept*(vecPWall/normC(vecPWall))
593.
594.                             #FWVect = alpha*vecTWall + beta*vecPWall
595.                             #FWVect = (FWVect/normC(FWVect))
596.
597.                             #Send both Follow Wall and Avoid Obstacles Vector
598.                             #l_Vect.acquire()
599.                             #FWDummy, AODummy = q_Vect.get()
600.                             #q_Vect.put([FWVect,AODummy])
601.                             #l_Vect.release()
602.
603.                             #print 'Time LIDAR ellapsed this run:{0}'.format(time.time()-t1)
604.                             #t1 = time.time()
605.
606.             except KeyboardInterrupt:
607.                 print 'Interrupt received in wallDetector!'
608.
609.                 try:
610.                     with Sweep(lidDev) as sLid:
611.                         sLid.stop_scanning()
612.                         sLid.set_motor_speed(0)
613.                 except:
614.                     print 'No LIDAR, still exiting.'
615.
616.
617.             finally:
618.                 print 'Ending wallDetector!'
619.
620.
621.     def main():
622.             '''************************************************************************************************
    ************************************************************'''
623.             #Las entradas son "conexion a Pixhawk" "conexion a LIDAR" "Ratio de giro de
    motor" "Ratio de muestreo de LIDAR" "MAX numero de puntos para reconstruir pared"
624.             #Por ejemplo, se puede correr este ejemplo utilizando "python
    SAFirstTestComm.py /dev/ttyUSB0 /devttyUSB1 4 500 7". Cambiar /dev/ttyUSBX por
    '127.0.0.1:14550' por ejemplo en la conexion de Pixhawk si se utiliza el software SITL de
    droneKit.
625.             '''************************************************************************************************
    ************************************************************'''
626.             ##################################################JJ SAVELOGS
    ##########################################################################
    ##################################
627.             # jindex=0
628.         ##########################################################################
    ##########################################################################
    ##END JJ#############
629.             if len(sys.argv)<6:
630.                 sys.exit('Inputs are "quadDevice", "LIDAR device", motorHz, sampleHz
    and numberOfLIDARPoints')
```

```
631.
632.            quadDev = sys.argv[1]
633.            lidDev = sys.argv[2]
634.            motorHz = int(sys.argv[3])
635.            sampleHz = int(sys.argv[4])
636.            numLIDAR = int(sys.argv[5])
637.
638.            q_Dist = Queue()
639.            q_Vect = Queue()
640.            l_Dist = Lock()
641.            l_Vect = Lock()
642.
643.            #Two peocesses are initiated. wD_P = Wall detector process. dM_P = Decision
    making process.
644.            wD_P = Process(target=wallDetector, args=(lidDev,motorHz,sampleHz,numLID
    AR,q_Dist,q_Vect,l_Dist,l_Vect,))
645.            dM_P = Process(target=decisionMaker, args=(quadDev,q_Dist,q_Vect,l_Dist,l_
    Vect,))
646.
647.            wD_P.start()
648.            dM_P.start()
649.
650.            #Will allow ctrl+c interruption to be heard by each process created.
651.            def signal_handler(signum,frame):
652.                global main_Interrupt
653.                main_Interrupt = True
654.                #print 'Program Finished (Main).'
655.
656.            signal.signal(signal.SIGINT, signal_handler)
657.
658.            while not main_Interrupt:
659.                time.sleep(0.1)
660.
661.            print 'Waiting for processes to join...'
662.
663.            #Waits for all processes to end.
664.            wD_P.join()
665.            dM_P.join()
666.            print 'All finished!'
667.
668.    main()
```